Toolset Reference Manual

SGS-THOMSON
MICROELECTRONICS

May 1995

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

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 487 00

Contents overview

Appendices

Contents

Appendices 299

Preface

About this manual

This manual is the *Toolset Reference Manual* and it is designed to cover the following products:

- ST20 toolset;
- ANSI C toolset;
- occam 2 toolset;

for the following hosts:

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

This manual provides reference material for each tool in the toolsets including command line options, syntax and error messages. Many of the tools in the toolset are generic to several toolset products e.g. the ST20 toolset, the ANSI C toolset and the occam 2 toolset and the documentation reflects this. Examples are given in C. The appendices provide details of toolset conventions, processor types, the ANSI C assembler, memory configuration files and the configuration languages.

A list of the tools supported by your particular toolset is given in the '*User Guide*' which accompanies your toolset. References in the documentation to tools, languages or processor targets which do not apply to your toolset, should be ignored.

About the toolset documentation set

The toolset documentation set comprises the following volumes:

User Guide

Toolset Reference

Language and libraries Reference

- *Toolset User Guide*

  Describes the use of the toolset in developing programs. The main stages of the development process are described and a '*Getting started*' tutorial is included. The '*Advanced Techniques*' section is aimed at more experienced users.

- *Toolset Reference Manual* (this manual - see above)

- *ANSI C Language and Libraries Reference Manual*

  The manual is divided into two parts: the toolset libraries and the language reference. A set of appendices is also provided. The libraries section lists the runtime library functions and provides detailed information about each function. A chapter also describes how to modify the runtime startup system by removing segments not required by the user's application. Only very experienced users should attempt this. The language reference for the toolset includes implementation and compliance data.

Other documents

Other documents provided with your toolset product include:

- *Delivery manual*

  This document gives installation data and is host specific.

Documentation conventions

The following typographical conventions are used in this manual:

Bold type Used to emphasize new or special terminology.

`Teletype` Used to distinguish command line examples, code fragments, and program listings from normal text.

Italic type In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.

Braces { } Used to denote optional items in command syntax.

Brackets [ ] Used in command syntax to denote optional items on the command line.

Ellipsis ... In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.

| In command syntax, separates two mutually exclusive alternatives.

Tools

1 icc - ANSI C compiler

This chapter describes in detail the ANSI C optimizing compiler icc. It describes the command line syntax, compiler options, preprocessor directives, optimization and other features of the compiler such as support for transputer code. The chapter ends with a list of error messages. Examples of the types of optimization supported are given in the appendices.

1.1 Introduction

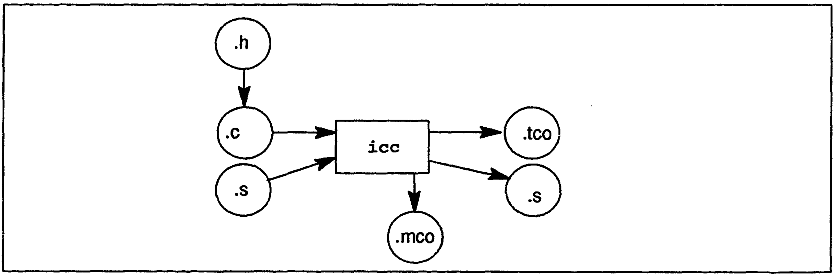
The ANSI C compiler conforms fully with the X3.159–1989 ANSI standard for the C programming language. This standard has now been ratified as "ISO/IEC 9899:1990 Programming languages — C". The ANSI C compiler provides support for concurrent programming as well as some additional extensions to the C language including compiler directives, pragmas and low level programming.

The ANSI standard for the C language defines the language including runtime library support, new types and function prototyping. The ANSI C compiler includes support for parallel programming through a set of library functions with associated types and structures, a mechanism for incorporating transputer code sequences, and a group of compiler pragmas for enabling compiler options in sections of code and for conveying directives to the linker. The transputer code mechanism supports the full set of transputer instructions and operations and also supports labels.

Parallel processing is achieved through a library of process, channel, and semaphore functions and their related types and data structures. Calls to the functions are compiled by icc into highly efficient parallel code for the transputer.

icc generates code for a particular transputer, transputer type, or class, and a target should be specified for all compilations.

The operation of the compiler in terms of standard toolset file extensions is shown below.



1.2 Running the compiler

To invoke the compiler use the following command line:

```
▶   icc   filename   {options}
```

where: *filename* is the C program source code. If no extension is given `.c` is assumed. Only one filename may be given on the command line.

options is a list of options given in table 1.1. Options to select the transputer target for the compilation are listed in appendix B.

> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.
>
> Options may be supplied in an indirect argument file, prefixed by '@'. See section A.1.2 for details.

If no arguments are given on the command line brief help information is displayed; the full help page is displayed by using command option '`HELP`'.

Note: `icc` must be invoked in a writeable directory, that is, one in which you (or any alias you use to invoke the compiler) have *write* access.

Option	Description
Transputer type	See appendix B for a list of options to specify transputer type.
`AS`	Assemble the input file to produce an object file. The compiler phase is suppressed. See section 1.2.6.
`COMPACT`	Produce an object file which may be compacted by the linker. *Compiler default.* See section 1.2.11.
`D` *symbol* `D` *symbol=value*	Defines a symbol. Same as `#define` *symbol* 1 at the start of the source file. Defines a symbol and assigns a value. Same as `#define` *symbol value* at the start of the source file.

`EC`
`EP`
`EZ`
	Disables checks for invalid type casts. ANSI compliance check.
	Disables checks for invalid text after `#else` or `#endif`. ANSI compliance check.
	Disables checks for zero-sized arrays. ANSI compliance check.

`FC`	Change the signedness property of plain `char` and plain bit-fields to be signed. The default is to compile as unsigned.
`FEEDBACK` *feedbackfile*	A *feedbackfile*, generated by the INQUEST profiling tool, is used as input to the compiler to improve the quality of optimization. See section 1.3.7.
`FH` `FM`	Performs a number of software quality checks. See section 1.2.9. Generates warning messages on `#defined` but unused macros.
`FS`	Directs the compiler to treat right shifts of signed integers as arithmetic shifts. See section 1.2.8.

Option	Description
FSA *number*	Changes the alignment for structs and unions to *number*. See section 5.1.3 of the '*ANSI C toolset language and libraries reference manual*'. **Note:** code compiled with the FSA option (whose interface contains structures whose alignment is changed by the FSA option) should not be mixed with code compiled without it, or with a different value for the FSA option.
FSC	Provides information on how the compiler has treated routines with respect to side effects.
FV	Reports all externally visible functions and variables which are declared but un-referenced, and have file scope.
G	Generates comprehensive debugging data. The default is to produce minimal debugging data. Debugging data is required for the correct operation of the debugging tools.
HELP	Displays full help information for the tool.
I	Displays detailed progress information at the terminal as the compiler runs.
J *dir*	Adds *dir* to the list of directories to be searched for source files incorporated with the #include directive in extended search paths. See section 1.5.9 for details.
KP	Inserts run-time code to check that pointers are correctly aligned, and that NULL pointers are not de-referenced. See section 1.2.10 for details.
KS	Inserts run-time code to check that the stack does not overflow. See section 1.2.10 for details.
NOCOMPACT	Produces an object file which cannot be compacted by the linker. See section 1.2.11.
O *outputfile*	Specifies the name of the output object file. If no filename is given the compiler derives the output filename from the input filename stem and adds the .tco extension.
O0	Disable optimization.
O1	Enable local optimization. *Compiler default.*
O2	Enable both global and local optimization.
P *mapfile*	Produces a map of workspace for each function defined in the file, and a map of the static area of the whole file. The map is written to the file *mapfile*. See section 1.4. **Note:** if this file is to be used as input to imap, it must be given an extension of the form: .mxx. The characters 'xx' are determined by the 2nd and 3rd characters of the extension given to the compiler object file. For example if the compiler object file takes the default extension .tco, the information file is given the extension .mco.
PG PL	Inserts instructions in the object code to collect data for '*call-graph*' profiling with the INQUEST profiling tool. Inserts instructions in the object code to collect data for '*line*' profiling with the INQUEST profiling tool.
PP	Runs the preprocessor and then terminates. The preprocessed source file is sent to stdout. Compilation is suppressed. See section 1.2.7.
PPC	Runs the preprocessor and then terminates. The preprocessed source file is sent to stdout. Compilation is suppressed. Comments are preserved in the preprocessed output. See section 1.2.7.
PR	Inserts instructions in the object code to collect data for '*routine*' (i.e. function) profiling with the INQUEST profiling tool.
OS	Optimize for space.
OT	Optimize for time. *Compiler default.*

Option	Description
S	Compiles the source file to assembly language and writes it to a file. Assembly is suppressed and no object code is produced. The file is named after the input file and given the `.s` extension.
U *symbol*	Disables a symbol definition. Equivalent to **`#undef`** *symbol* at the start of the source file.
W	Suppress all warning messages.
WA	Suppresses messages warning of '=' in conditional expressions.
WD	Suppresses messages warning of deprecated function declarations.
WF	Suppresses messages warning of implicit declarations of `extern int()`.
WN	Suppresses messages warning of implicit narrowing or lower precision.
WS	Suppress warning messages about possible side effects.
WT	Suppresses messages warning of the possibility of less efficient code when compiled for a transputer class. **Note:** this option no longer has any effect and the warning messages are no longer generated.
WTG	Suppress messages warning of trigraphs.
WV	Suppresses messages warning of non-declaration of `void` functions

Table 1.1 Standard `icc` compiler options

Examples of use:

```
icc –st20 hello.c
ilink –st20 hello.tco –f cstartup.lnk
icconf  hello.cfs
icollect  hello.cfb

icc –t450 hello.c
ilink –t450 hello.tco –f cstartup.lnk
icconf  hello.cfs
icollect  hello.cfb

icc –t805 hello.c
ilink –t805 hello.tco –f cstartup.lnk
icconf  hello.cfs
icollect  hello.cfb
```

1.2.1 Input/output files

The compiler command line will accept the following types of input file:

- A C source file, which by convention is given a '`.c`' file extension. Although any filename, which is legal on the host system, will be accepted. If a source file is specified without an extension, the compiler will assume it is a C source file and search for a file with a '`.c`' file extension.

- An assembler source file, which by convention is given an '`.s`' file extension. Assembler source files are input to the compiler's built-in assembler, using the '`AS`' option, which suppresses compilation. **Note:** a file extension (other than '`.c`') *must* be specified. Even though the '`AS`' option is specified to invoke the assembler, the compiler will assume a '`.c`' file extension if a file extension is not specified on the command line.

Source files may contain preprocessor directives. If an assembler source file contains preprocessor directives then it must be input to the compiler, using the 'preprocess-only' '`PP`' or '`PPC`' command line options. The output from the preprocessor may them be input to the assembler as described above.

The compiler produces a single 'primary' output file and optionally a 'secondary ' output file. The type of output file produced depends on the command line options used. The name of a primary output file is specified using the '`O`' command line option. Primary output files are generated as follows:

- By default the compiler generates an object file in *T*ransputer *C*ommon *O*bject *F*ile *F*ormat (TCOFF). Object files are required to be in this format to be compatible with other tools in the toolset. If the '`O`' command line option is not specified, the input filename is used and a '`.tco`' extension is added.
- An assembler source file. This is generated using the '`S`' command line option, the assembler is suppressed. If the '`O`' command line option is not specified, the input filename is used and an '`.s`' extension is added.
- A text file generated by using the 'preprocess-only' '`PP`' or '`PPC`' command line options. If no output file name is specified, then the output is written to `stdout`.

Figure 1.1 summarizes `icc`'s input and primary output file options. Sections 1.2.6 and 1.2.7 describe the use of the assembler and preprocessor options in further detail.

An optional secondary output file may also be produced by specifying the '`P`' command line option together with a filename. This generates a map file describing the code and data layout of the object file. Naming conventions for map files are described in table 1.1. Section 1.4 describes the format of the map file.

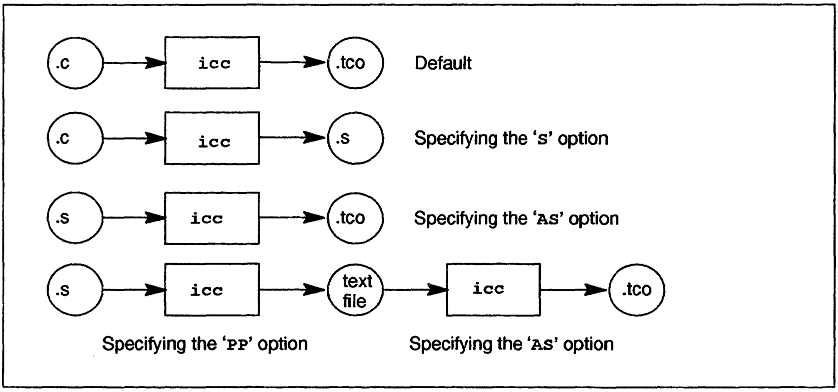

.c
icc
.tco
Default
.c
icc
.s
Specifying the 'S' option
.s
icc
.tco
Specifying the 'AS' option
.s
icc
text file
icc
.tco
Specifying the 'PP' option
Specifying the 'AS' option

Figure 1.1 `icc` input/output file options

1.2.2 Transputer targets

The compiler generates code for a specific transputer type. This means that a processor type must be specified for all transputer targets. The only time a processor type does not have to be specified is when the preprocessor is selected, see section 1.2.7.

Transputers are also grouped into classes for the purpose of generating common code suitable for running on a number of different transputer targets. Transputer classes group transputers according to word size and instruction set compatibility. They can be used to generate code for combinations of transputers.

The use of transputer types and classes in developing programs is explained in appendix B. The command line options for selecting a transputer target are given in this appendix.

1.2.3 Error modes

All code in mixed language transputer programs must be compiled and linked in the same or a compatible error mode. `icc` always generates code in UNIVERSAL error mode, which is compatible with HALT and STOP error modes created by other SGS-THOMSON compiler toolsets.

The error mode for a mixed language program can be consolidated into a single mode for the entire program by specifying the appropriate linker option. If no mode is specified the linker generates the program in HALT mode.

1.2.4 Default command line options

Commonly used command line parameters can be defined in the host environment variable `ICCARG`. Parameters specified in this way are automatically added to the start of the command line when the compiler is invoked.

Command line parameters must be specified in `ICCARG` using the syntax required by the `icc` command line.

1.2.5 Search paths

The search rules are described in appendix A. Search path rules 1 and 2 are used for locating files specified on the command line.

Search paths for files imported with the `#include` compiler directive differ slightly from those for files specified on the command line and can be extended by the use of special syntax and a command line option. Details of this facility can be found in section 1.5.9.

1.2.6 Using the assembler

Assembler source files may be assembled by using the `icc` command line option '`AS`'. This causes the compilation phase of the compiler to be suppressed and the input file to be passed directly to the assembler. If the input assembly source file contains preprocessor directives, the compiler preprocessor must first be used to process the source file; the output from the preprocessor may then be used as input to the assembler.

Note: the compiler also has an option 'S' which will compile a C source file into an assembler source file. The assembler phase is suppressed.

The use of the assembler is described in appendix D, together with examples of how it is invoked. The file name conventions for assembler files and the command options which may be used with the assembler are listed. The appendix also describes the syntax of assembler directives and lists the error messages which may be generated by the assembler.

1.2.7 Using the compiler preprocessor

Source and header files may be preprocessed using the 'PP' command line option. The preprocessor implements translation phases 1 to 4 of the ANSI Standard, section 2.1.1.2. The 'PPC' option additionally preserves comments in the preprocessed output. These two options suppress compilation and can be thought of as 'preprocess-only' mode.

The preprocessor is used to resolve preprocessor directives (i.e. directives with a '#' prefix) and to perform macro expansion. Preprocess-only mode is particularly useful for header files or for assembler source files because the assembler does not have the ability to process preprocessor directives. The output from preprocess-only mode may then be fed back into the compiler or assembler for further processing.

In preprocess-only mode, a target processor type need not be specified. In this case the preprocessor symbol _PTYPE takes the value zero '0' which is a dummy value. If a target processor is specified then _PTYPE will be set as normal.

Preprocess-only mode generates a text file which by default is sent to standard out (stdout), alternatively the 'O' command line option can be used to name an output file.

1.2.8 Compatibility with other C implementations

A number of compiler options are provided which may assist users porting existing C code to transputer systems.

Arithmetic right shifts

By default, the compiler implements right shifts of signed integers as logical shifts, the command line option FS switches the implementation. This allows correct working of programs which assume that right shifts of signed values propagate the sign.

Signedness of char

By default the compiler implements plain chars as unsigned chars. The command line option FC switches the implementation to signed char, plain bit-fields are also signed. Details of type representation are given in chapter 5 of the '*ANSI C Language and Libraries Reference Manual*'.

Alignment of structs/unions

By default, the compiler aligns structs and unions on a word boundary. The command line option 'FSA *number*' modifies this. Chapter 5 of the '*ANSI C Language and Libraries Reference Manual*' describes the use of this option in more detail.

Some C implementations align a struct/union according to the strictest alignment requirements of the fields of the struct/union; this can be achieved using the 'FSA1' option.

1.2.9 Software quality check

The FH option allows policing of software quality requirements. The option requires all externally visible definitions to be preceded by a declaration (from a header file), thus guaranteeing consistency.

When the FH option is used the compiler reports:

- all forward static declarations which are unused when the function is defined.
- all repeated macro definitions (this is when macros are redefined to the same value; redefining a macro to a different value is always diagnosed as an error).

1.2.10 Runtime checking options

The KP and KS command line options cause the compiler to insert run–time code to perform checking.

Enable pointer dereference checks

When the KP option is specified, the compiler inserts a check each time a pointer is dereferenced. This check ensures that the pointer is not NULL and that the pointer is correctly aligned for the type of object being accessed. For example, in the following code,

```
int *pi;
char *pc;
*pc = (char)(*pi);
```

two pointer checks will be inserted: one to check that pi is not NULL and that it points to a word–aligned object, and another to check that pc is not NULL.

Note: that no check is inserted if a pointer is assigned or read but not dereferenced. For example, no checks will be inserted in the following code,

```
int *pi1, *pi2;
pi1 = pi2;
```

Pointer dereference checks are not performed in functions which have been marked with the pragma IMS_nolink.

Enable stack checks

When the KS option is specified, the compiler inserts a check on entry to each function. This check ensures that the stack has enough space available for the function's workspace, plus a margin for calling library functions which do not contain stack checks. This margin is currently 150 words.

As the stack check always ensures that there is a margin free below a function's workspace, any leaf functions (functions which do not call other functions) whose workspace fits into this margin do not require a stack check. The compiler will automatically suppress the stack check for such functions.

Stack checks are not performed in functions which have been marked with the pragma IMS_nolink.

1.2.11 Compactable code

By default the compiler will generate code 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 which the linker then uses to determine the optimal length of the instruction. The object file produced by the compiler will be larger than if compactable output was switched off, using the icc 'NOCOMPACT' option, but after linking the code will be smaller and faster.

Compacted and non-compacted code may be mixed, however, non-compacted code may not be supported by future toolsets.

1.3 Compiling with optimization switched on

Compiling source code with optimization enabled generates highly efficient object code. The purpose of optimization is to improve the execution time of object code as well as the program's use of memory i.e. workspace or stack or code-size. Compiler options QT and QS enable the user to control whether the optimization performed is predominantly to improve execution time or memory use. Optimization does not affect the functionality of the program, although compile times will be slower when a high level of optimization is performed.

The compiler implements both local and global levels of optimization:

- The global optimizations include: common subexpression elimination, strength reduction, loop invariant code motion and tail–call optimization. The optimizer examines each function as a single unit, enabling it to obtain as much information as possible about that function, while performing the optimization. Global optimization is more complex than local optimization; generally the more information available to the optimizer the better chance the optimizer has of improving code. The compiler pragma IMS_nosideeffects enables the user to clarify the behavior of individual functions. (Compiler pragmas are described in section 1.5.11).

- The local optimizations include: flowgraph, peephole and redundant store elimination. To perform these optimizations efficiently, the optimizer only needs to operate on short sequences of code.

1.3.1 Advantages of enabling optimization

The advantages of enabling the optimizing features of the compiler are that:

- It saves development time by relieving users of the need to optimize their code themselves.
- It allows users to write more readable, and hence more maintainable, code because they can rely on the compiler to transform the code into a more efficient form.
- There are some optimizations which cannot be performed by the user at a source code level but can only be performed by the optimizing compiler at compile time.
- The compiler is able to analyze the cost/effectiveness of potential optimizations and will only apply an optimization where a saving can be made in either execution time or space.

1.3.2 When optimization should not be used

If it is required to debug a program it may be wise to disable optimization using the 'O0' command line option. The compiler will generate debug information requested via the 'G' option when optimization is enabled, however, the SGS-THOMSON debugging products will produce more accurate results if optimization is disabled in the compiler.

1.3.3 Optimization options

Disable optimization O0

The option 'O0' disables all optimization which can be specifically enabled at the command line using the 'O1' and 'O2' options.

Enable local optimization O1

This option is enabled by default and applies the following local optimizations:

- Flowgraph optimization, including dead code elimination.
- Peephole optimization.
- Redundant store elimination.

Local optimizations are described, with examples, in section C.1.

In addition, workspace allocation by coloring is enabled. This is in fact a global optimization and is described in section C.2.6.

Enable local and global optimization O2

This option, enables the following local and global optimizations:

- All optimizations enabled by option O1.
- Global common subexpression elimination.
- Loop invariant code motion.
- Strength reduction.
- Tail-call optimization.
- Tail recursion optimization.

Global optimizations are described, with examples, in section C.2.

Optimize for time QT

This option controls how optimization is applied once it has been enabled by either the O1 or O2 options. The option instructs the compiler to perform only those optimizations which will not reduce the speed of the program. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate the faster code. This option is enabled by default.

Optimize for space QS

As above, but does the reverse, i.e. only performing those optimizations which will not increase the size of the program. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate the more compact code.

There is no definitive list for either the QT or QS options, as to which optimizations will or will not be applied. This will vary depending on the code being optimized.

1.3.4 Enable side effects information messages

The FSC option enables the generation of information messages about the 'side effect' characteristics of functions as the compiler performs optimization. The messages report the actions of the compiler to give the user visibility of how functions are treated with respect to side effects. The messages are purely informational and do not signal any required response from the user.

Information messages are listed in section 1.8.4. Side effects are discussed in more detail in section 1.5.11, as part of the description of the compiler pragma IMS_nosideeffects.

1.3.5 Disable side effect warning messages

The WS option disables messages warning users that functions marked as side effect free may in fact still cause side effects. See section 1.5.11.

1.3.6 Language considerations

Before the compiler can optimize a function call it has to be sure that the optimization is safe i.e. that it will not break the code. Therefore it will treat function calls with caution, assuming that they may modify global variables, unless it can deduce with certainty their true behavior.

This section outlines language features which affect the implementation of optimization by `icc`.

`const` keyword

The `const` keyword states that after it is initialized, a variable cannot subsequently be modified by the program. For a `const` variable, the compiler does not have to make worst case assumptions about its being modified when ambiguous modifications are seen. If a variable is never modified, then declaring it as `const` will, in general, allow the compiler to do a better job of optimizing.

Note: when pointers to `const` objects are used e.g.

```
const char *p
```

the `const` keyword does not guarantee that the `char` will not be modified, just that it will not be modified through pointer `p`.

`volatile` keyword

The `volatile` keyword states that a variable may change asynchronously, or have other unknown side effects. The compiler will not move or remove any loads or stores to a volatile variable. `volatile` should be used for variables shared between parallel threads (or variables modified by interrupt routines), or variables which are mapped onto hardware devices.

`register` keyword

The `register` keyword is taken as a hint to the workspace allocator to allocate the variable at a small workspace offset.

1.3.7 How to use feedback information to improve optimized code quality

When the compiler is optimizing code, it requires information about how the code executes: for example, it needs to know how many times a loop will iterate, or given a conditional branch, whether the branch is likely to be taken or not. This information is generally not available to the compiler, so it has to make educated guesses, which enable it to achieve a general level of optimization without perhaps 'fine-tuning' performance in particular circumstances. One way of improving the compiler's ability to optimize code is to actually gather the required information during execution of the program, and then rerun the compiler, feeding in the gathered information. Thus, the compiler can be 'taught' how the program executes. If the information given to the compiler accurately represents how the program normally executes, then the quality of code generated can be improved.

This method of program development is supported by the combined use of the ANSI C toolset optimizing compiler and the profiling tools supplied in the INQUEST product.

The sequence of events required is:

Step 1 Compile the program with global optimization and line profiling enabled (using the compiler's 'O2' and 'PL' options). If you are only worried about one or two critical routines then you can just enable profiling information for the source files with those routines in.

Step 2 Link, configure, and collect the program as usual.

Step 3 Execute the program with a sample input. **Note:** that it is important that the sample input is an accurate representation of the program's normal input, so that the behavior of the program on this sample is close to the normal behavior of the program. (Alternatively, for embedded systems, it may be the behavior in some critical situation which is most important, so it is required that the compiler should optimize the code for this critical situation, in which case, the sample input should represent this critical situation.)

Step 4 After the program terminates, run the profiling tool of the INQUEST product to extract '*line*' profiling information from the target network and create the '*feedback*' files, which contain the information the compiler needs. (Currently only '*line*' profiling information can be used to generate '*feedback*' files).

It is possible to execute the program on a number of different sample inputs, and accumulate the feedback information. To do this, repeat steps (3) and (4) for each sample input, and use the INQUEST '*line*' profiling options to append new data to the 'feedback' files, rather than recreating them each time.

Step 5 Now recompile the source files for which you have gathered profiling information, specifying the feedback file (using the compiler's 'FEEDBACK *filename*' option). Do not enable any profiling options in this recompilation.

Note: that the source code must not have changed in any way between the compilation with profiling enabled and the compilation using the feedback files, otherwise the compiler will not be able to work out which program constructs, the feedback information refers to. If the compiler detects that the source has changed, it will report the error:

```
Serious-icc-<filename>(<line>)- Feedback file is out of step with source
```

Step 6 Relink, configure and collect the program as usual.

Example:

Suppose you have a program made up of two files `bit1.c` and `bit2.c` and you are compiling for an ST20 target:

Step 1: compile with line profiling enabled:

```
icc bit1.c -st20 -O2 -pl
icc bit2.c -st20 -O2 -pl
```

Global optimization is enabled by '`O2`'.

Step 2: link, configure and collect:

```
ilink bit1.tco bit2.tco -f cstartup.lnk -st20 -o bit.lku
icconf bit.cfs
icollect bit.cfb
```

Step 3: execute the program, for example:

```
irun -sb bit.btl
```

Chapter 15 of this manual has further details about `irun` and the accompanying '*User Guide*' describes how to load and execute programs.

Step 4: gather profiling information using the INQUEST *line* profiling tool, and create a feedback file, for example:

```
iline bit.btl -feed
```

The profiler will create two feedback files called `bit1.d` and `bit2.d`.

Consult the profiler documentation which accompanies your INQUEST product for details of how to use the *line* profiling tool.

Step 5: recompile your program using the feedback files (generated by step 4):

```
icc bit1.c -st20 -O2 -feedback bit1.d
icc bit2.c -st20 -O2 -feedback bit2.d
```

Step 6: relink, configure and collect:

```
ilink bit1.tco bit2.tco -f cstartup.lnk -st20 -o bit.lku
icconf bit.cfs
icollect bit.cfb
```

1.4 Memory map

The compiler may be instructed, via the `P` *mapfile* option, to produce a map of workspace for each function defined in the file, and a map of the static area of the whole file. The file contains information which may assist the user during program debugging and can be used as input to the memory mapper **`imap`**. The map is written to the file *mapfile*.

The file consists of a series of workspace maps; one for each routine, giving details of workspace requirements. These are followed by a series of section maps; one for each section of code or data, listing details of its contents.

The file is generated in text format and is structured as follows:

- The name of the source file for which the map of code and data is being produced.
- Version data for the compiler.
- The target transputer of the compilation, T805, T400 etc.
- The error mode of the compilation, this is always UNIVERSAL for C programs.
- Name of the routine for which the map of workspace is being produced. Items in the workspace map are given in ascending order of workspace offset, expressed as bytes or words as indicated in the heading. If the workspace map is in words, individual items may still be expressed and annotated as byte offsets. The following information is provided:
  - A list of local variables giving their offset into the routine's workspace. This list may include temporary variables introduced by the compiler.
  - A list of formal parameters giving their name and offset into the routine's workspace. Parameters added by the compiler may also be listed, see table 1.2 for a list of their names (further details of these parameters can be found in section 5.16 of the '*ANSI C Language and Libraries Manual*').
  - The workspace requirement of the routine. **Note:** this includes the four word call overhead introduced by the transputer *call* instruction.
- Name of the section for which the section map is being produced. Items in the section map are given in ascending order of section offset.
  - A list of static variables or routines, giving the following details:
    - Name of static variable or routine. This may be in the form '*<name>*`%xp`', see table 1.3
    - Type of variable or routine
    - Offset in bytes into static data or code area
    - Other properties of variable or routine, see table 1.3.

Static variables are either placed in the static or code areas. Details of how the compiler allocates space for static data are given in section 5.15 of the '*ANSI C Language and Libraries Reference Manual*'.

```
Map of code and data for source file twoprocs.c
===============================================
Created by INMOS C compiler  Version  4.01.09 (02:24:52 Mar 11 1995)  (SunO
S-Sun4)

Target processor : ST20
Error mode       : UNIVERSAL

Map of workspace
----------------
Routine : hello_proc

No local variables

Formal parameter name           Offset (bytes)

<return_address>                0
<gsb>                           4
p                               8

Workspace size = 16 bytes

Map of workspace
----------------
Routine : world_proc

No local variables

Formal parameter name           Offset (bytes)

<return_address>                0
<gsb>                           4
p                               8

Workspace size = 16 bytes

Map of workspace
----------------
Routine : main

Variable name                   Offset (bytes)

<compiler_temporary>            0
hello                           4
world                           8

Formal parameter name           Offset (bytes)

<return_address>                12
<gsb>                           16

Workspace size = 28 bytes

Section map
-----------
Section name : text%base : size = S:8 bytes

Name                            Type        Offset (bytes)

hello_proc                      code        0          global
world_proc                      code        S:9        global
main                            code        S:10       global
```

Figure 1.2   Example compiler map

Formal parameter
Compiler temporary
Result pointer
Return address
Global static base pointer (gsb)
Static link

Table 1.2 Parameters inserted by compiler

Property	Description
global	Globally visible static item.
static	Static item which is not globally visible.
pointer to external object	Static item introduced by the compiler to enable code to access an external object. The name of the external object is used as the prefix to the compiler generated name. e.g. '`fred%xp`' is a static item introduced by the compiler which points to an external object named '`fred`'.
translated from data name	Static items whose name has been modified by the `IMS_translate` pragma are listed under the name that is put into the object file. They are annotated with the message: '`translated from` *sourcename*', where *sourcename* is the name used in the source file.

Table 1.3 Static variable properties

1.4.1 Notes on the compiler memory map format

1 The message "`No local variables`" may be displayed if no user variables are found, however, compiler temporaries may have been assigned to workspace.

2 If a file does not contain static data, such information will not be present in the map file and in this case the '`static%base`' section map will not appear. The compiler does not generate an explicit "`No static data`" message.

3 When optimization is enabled using the command options `O1` or `O2` the compiler may remove some local variables from the map file generated. In such cases the variable will not occupy any space in workspace. These variables are listed at the end of the local variable map together with the message "**Not Allocated**".

4 Some offsets may only be resolved by the linker. In this case the compiler inserts the string '`S:`*n*', which informs the linker there is a local symbol '*n*' which needs resolving. The linker inserts the value of the symbol in a `LOCALVALUE` record in the linker map file and this value will be inserted in the map file generated by `imap`. See chapters 10 and 13.

5 Variables and external symbols are truncated to a maximum of 32 characters.

6 Information generated in the compiler map file may be extracted by the `imap` tool. This tool can be used to produce a memory map for the program after it has been compiled, linked and collected. See chapter 13.

1.5 Compiler directives

1.5.1 #define

Syntax: `#define` *name* [(*arg1,. . .,argn*)] [*value*]

`#define` allows simple macro substitution to be performed. In its simplest mode of operation *name* and *value* represent a series of ASCII characters causing the preprocessor to substitute all occurrences of *name* by *value* (which may be null). Arguments may also appear after the name, and when this happens the preprocessor will still replace all occurrences of *name* and its following arguments by *value*, but in this case the value string will have been defined in terms of the expected arguments, and will therefore exhibit a dependence on the original text.

```
#define YES 1       /* replace all occurrences
                       of YES by 1 */

#define max(a,b) ((a) > (b) ? (a) : (b))
 /* max(2,4) will be replaced by
    ((2) > (4) ? (2) : (4)) */
```

1.5.2 #elif

Syntax: `#elif` *constant_expression*

This directive can be used in place of the sequence

```
#else
#if constant_expression
```

1.5.3 #else

Syntax: `#else`

This directive can be used with the `#if`, `#ifdef`, and `#ifndef` directives to mark the beginning of text which will be ignored whenever the expression following the `#if` evaluates to a non-zero value.

1.5.4 #endif

Syntax: `#endif`

This directive must be used with the `#if`, `#ifdef`, and `#ifndef` directives to mark the end of the text which may be affected by the `#if . . . #else . . . #endif` construct.

1.5.5 #error

Syntax: `#error` *text*

This directive causes an explicit error with the text following the directive displayed in the error message. This is useful for determining which pieces of code are being bypassed by a construct of the form `#if` ... `#else` ... `#endif`.

1.5.6 #if

Syntax: `#if` *constant_expression*

This directive, along with the `#else` and `#endif` directives, is used in a similar way to the if . . . else construct of many high level programming languages. When it is encountered, the preprocessor evaluates the following constant expression and if it is zero it ignores all text up to the following `#else` or `#endif` directive. If, however, the expression evaluates to non-zero, then the text between the `#else` and `#endif` directives (if any) is ignored. This mechanism would typically be used to allow conditional compilation.

As an extension to this directive, the preprocessor also allows 'if defined' type expressions. In this case 'defined' is used as a unary operator which returns true if its operand represents an identifier that is currently defined within the preprocessor's symbol table, and false if it is not. By combining this operator with the logical operators it is possible to build complex expressions e.g.

```
#if defined foo & ! defined dummy

    /* if foo is defined and dummy is not */
```

1.5.7 #ifdef

Syntax: `#ifdef` *identifier*

This directive works in a similar way to the `#if` directive, but instead of basing its decision on the result of an expression it uses the existence or non-existence of the identifier within the preprocessor's symbol table as the criterion. If the identifier has not previously appeared in a `#define` directive or if it is not one of the predefined identifiers then all text up to the following `#else` or `#endif` directive is ignored; otherwise all text between the `#else` and `#endif` directives is ignored.

1.5.8 #ifndef

Syntax: `#ifndef` *identifier*

This directive is similar to `#ifdef`, except that the text is passed if *identifier* is not currently defined.

1.5.9 `#include`

Syntax: `#include` *filename*

The `#include` directive instructs the preprocessor to read the contents of the named file as if they were at the current position in the current file. The filename must be enclosed within angle brackets (<*filename*>) or double quotes ("*filename*"). The two forms generate different search strategies.

If angle brackets are used *only* those directories specified by `ISEARCH` are searched. No other directories (including the current directory) are searched. This method is mainly used to include the standard library header files.

If double quotes are used, then the compiler first searches for the file in the directory of the top level source file (relative directory names are relative to the directory of the current source file). If the file is not found the search continues with the list of directories specified after the compiler '`J`' option. If the file is still not found, or if no list is given, directories specified by `ISEARCH` are searched.

A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file. There is no fixed limit to `#include` nesting.

Relative directory names

Relative directory names are treated as relative to the directory containing the current source file.

Backslash character in filenames

In included filenames the backslash is not treated as introducing an escape sequence unless it is followed by another backslash ('\\').

1.5.10 `#line`

Syntax: `#line` *linenumber* [*filename*]

This directive instructs the compiler that subsequent lines begin with line number *linenumber* in the file *filename*. If no file name is specified, the original name is retained. *linenumber* must be within the range 1 to 32767 inclusive.

1.5.11 `#pragma`

Syntax: `#pragma` *pragma* (*params*)

This directive activates and deactivates various compiler options in sections of C code. It may be used to set (or override) options specified on the command line. Most pragmas also take parameters or numerical arguments. Table 1.4 lists the main compiler pragmas.

Option	Description
IMS_codepatchsize	Specifies the number of bytes reserved by the compiler for a linker code patch. Requires the compiler NOCOMPACT option to be used.
IMS_descriptor	Creates a TCOFF descriptor for C functions.
IMS_interrupt_handler	Causes the specified function to be treated as an interrupt handler function. Only applicable when compiling for a T450 or ST20 target. Should not be used in conjunction with the IMS_nolink pragma.
IMS_linkage	Enables the user to change the order in which code modules are linked together.
IMS_modpatchsize	Specifies the number of bytes reserved by the compiler for a linker module number patch. Requires the compiler NOCOMPACT option to be used.
IMS_nolink	Compiles a specified function without a global static base parameter. Should not be used in conjunction with either the IMS_interrupt_handler or the IMS_trap_handler pragmas.
IMS_nosideeffects	Marks the specified function as being side effect free.
IMS_off	Disables specific compiler actions.
IMS_on	Enables specific compiler actions.
IMS_place_at_workspace_offset	Allocates a local variable at a specific offset in the local workspace.
IMS_translate	The compiler replaces all references to a specified name with a new name.
IMS_trap_handler	Causes the specified function to be treated as a trap handler function. Only applicable when compiling for a T450 or ST20 target. Should not be used in conjunction with the IMS_nolink pragma.
Further details of each option is given below.	

Table 1.4 icc compiler pragmas

Pragma IMS_codepatchsize

Specifies the number of bytes reserved by the compiler for a linker code patch.

Syntax: `#pragma IMS_codepatchsize (n)`

n has a default value of 6 for 32-bit targets and 4 for 16-bit targets.

Note: this pragma has no effect when compactable object code is generated, which is the default. The compiler NOCOMPACT option must be specified on the command line.

Pragma IMS_descriptor

The pragma IMS_descriptor creates a TCOFF descriptor for C functions. It also causes the definition of two TCOFF symbols giving the workspace and vectorspace requirements of the function. This pragma is of particular use when modifying the C runtime startup code, further details of which are given in chapter 3 of the '*ANSI C*

Language and Libraries Reference Manual'. It is also applicable when making use of the dynamic loading facility provided in the C library (see chapter 2 of the '*ANSI C Language and Libraries Reference Manual*' and the 'Dynamic code loading' chapter of the '*User Guide*').

Syntax: **`#pragma IMS_descriptor`** (*function-name, language_type,* \
workspace, vectorspace, "descriptor–string")

The parameters to the pragma are given in table 1.5.

The rules governing the use of this pragma are as follows:

- The function must be externally visible.
- The function must have been declared before the pragma appears.
- The function must not have been defined before the pragma appears.
- The pragma must appear in the same file in which the function is defined.
- Only one descriptor pragma can exist per function.
- No argument to the descriptor pragma can be the result of earlier preprocessor substitutions.

functionname	Name of the C function to which the descriptor applies.

language_type	The language in which the descriptor string is written. The language is given as a keyword:
	`unknown`
	`occam`
	`ansi_c`
	`fortran`
	`iso_pascal`
	`modula2`
	`ada`
	`assembler`
	`occam_harness`
	Alternatively the *descriptor–string* may be an empty string, however, a language type must still be given.

workspace	The amount of workspace required by the function, expressed as a number of words.
vectorspace	The amount of vector space required by the function, expressed as a number of words. This is usually '`0`' for C functions.
"descriptor–string"	This is the descriptor string itself. If the string is not empty then it must contain an occam style function declaration equivalent to the C function prototype.

Table 1.5 Parameters to `IMS_descriptor`

An example of the use of this pragma follows:

```
void centry(int bill);

#pragma IMS_descriptor(centry, occam, 32, 0, \
                       "PROC centry(VAL INT bill)\n  SEQ\n:")

void centry(int bill)
{
  /* function body */;
}
```

This defines an occam descriptor for the function `centry`. A requirement for 32 words of workspace and no vectorspace is also recorded in the descriptor. The syntax for the descriptor string is the standard syntax for occam descriptors.

Note: type compatibility between the parameters in occam and C is retained by following the rules given in the 'Mixed language programming' chapter of the accompanying '*User Guide*' for those toolsets which support mixed language programming.

Example TCOFF output from the above can be obtained using the '`t`' option on the lister tool `ilist`, as follows:

```
00000080 SYMBOL EXP "centry" id: 4
...
00000092 SYMBOL EXP UNI "centry'ws" id: 5
0000009F SYMBOL EXP UNI "centry'vs" id: 6
000000AC DEFINE_SYMBOL id: 5 32
000000B1 DEFINE_SYMBOL id: 6 0
000000B6 DESCRIPTOR id: 4 lang: OCCAM
ws: 32 vs: 0
PROC centry(VAL INT bill)
  SEQ
:
```

Pragma `IMS_interrupt_handler`

This pragma indicates to the compiler that the named function is to be used as an interrupt handler.

Syntax: `#pragma IMS_interrupt_handler (`*function-name*`)`

The named function must have been declared before the pragma is encountered.

A warning is generated and the pragma is ignored if,

- the named function has been defined before the pragma is encountered;
- its parameter is not a function identifier;
- it is given no parameters or more than one parameter;
- the target processor is not a T450 or ST20.

Where support is provided by the toolset for interrupt handling, details of how to use the interrupt handling facilities are given in the accompanying '*User Guide*'.

Pragma `IMS_linkage`

Enables the user to change the order in which code modules are linked together; this may aid the use of faster on–chip RAM.

Syntax: `#pragma IMS_linkage ([`*"name"*`])`

The compiler creates the object code into a section named `"text%base"`. The `IMS_linkage` pragma causes the compiler to change the name of the section to that supplied in the string. If no string is present, `"pri%text%base"` is used; this section being inserted at the front by the linker in the default case. A linker directive (see 10.4.6) controls of the ordering of the sections.

The linkage directive should appear at the start of the code, before any function definitions.

Pragma `IMS_modpatchsize`

Specifies the number of bytes reserved by the compiler for a linker module number patch.

Syntax: `#pragma IMS_modpatchsize (`*n*`)`

n has default values of 3 for 32-bit targets and 2 for 16-bit targets.

Note: this pragma has no effect when compactable object code is generated, which is the default. The compiler `NOCOMPACT` option must be specified on the command line.

Pragma `IMS_nolink`

The pragma `IMS_nolink` compiles a specified function without a global static base parameter. The function must already have been declared but must not have been defined or called. This pragma is used for importing code written using languages such as occam which do not use static data, and for exporting C functions to the same languages.

Syntax: `#pragma IMS_nolink` (*function-name*)

The following code uses the pragma to allow an occam routine `OCCAMREALOP` to be called in a C program:

```
extern float OCCAMREALOP(const float x,
                         const int op,
                         const float y);
#pragma IMS_nolink (OCCAMREALOP)
 :
 :
 float x, y, z;
z = OCCAMREALOP(x, op_add, y);
```

The following code allows the C function max to be called from occam:

```
extern int max(const int x, const int y);
#pragma IMS_nolink (max)
extern int max(const int x, const int y)
{ return x > y ? x : y; }
```

Note: functions which have had the `IMS_nolink` pragma applied may not be called through a pointer. The library routine `call_without_gsb` is supplied to allow a call through a pointer to a nolinked function.

Pragma `IMS_nosideeffects`

This pragma enables the user to provide the compiler with more information about functions, whose behavior could otherwise be ambiguous and therefore helps to maximize the degree of optimization performed. Its syntax is as follows:

`#pragma IMS_nosideeffects (`*function-name*`)`

where: *function* must have been declared but not defined or instanced before the pragma is encountered. A warning is generated and the pragma ignored if:

- the named function has been defined, part defined or instanced before the pragma is encountered;
- its parameter is not a function identifier;
- it is given more than one parameter or none at all.

Functions which are marked as side effect free enable the compiler to avoid making worst–case assumptions about variables modified when the function is called.

A function is side effect free when, within the body of a function:

- there are no assignments to variables which are defined outside the function, including writing through a pointer where the pointer points to a variable defined outside the function e.g. I/O or 'passing by reference' is not allowed;
- there are no assignments to static variables (although initialization of static variables within definitions are allowed);
- there are no reads from or assignments to volatile variables;
- there are no calls to functions which may have side effects.

The user must ensure that a function meets the above criteria before marking it with the pragma `IMS_nosideeffects`. If a function is marked as side effect free and it does not meet the above criteria, the compiler will report an error. Assembler inserts i.e. `__asm` statements and pointers to functions, are a particular risk area, as the user might unconsciously introduce a function which does break the above criteria. For this reason, the compiler will warn the user, each time assembler code or a pointer to a function is encountered. **Note:** these warning messages may be disabled by using the WS command line option.

The compiler may mark a function as side effect free, even though the user has not applied the IMS_nosideeffects pragma. Use of the FSC command line option will cause an information message to be generated should this occur.

The compiler will only mark those functions that it detects as unambiguously side effect free and that are not already marked. It is possible for there to be functions which are side effect free that the compiler does not detect; for such functions, only the application by the user of the pragma IMS_nosideeffects will cause these functions to be marked as side effect free.

If a function that is side effect free is externally visible, then it may be useful to put the pragma IMS_nosideeffects into the header file following the prototype of the function. This enables the compiler to generate better code for calls to that function from other files.

Pragmas IMS_on and IMS_off

IMS_on enables specific compiler actions; IMS_off disables the same actions. Both pragmas take a list of parameters, (separated by commas) which specify the actions to be enabled or disabled, they are listed in table 1.6.

Syntax: #pragma IMS_off (*params*)
Syntax: #pragma IMS_on (*params*)

Parameter	Short form	Description
channel_pointers	cp	Treats a variable of type Channel in the scope of the definition typedef const volatile void * as a channel type for the debugger. Default is off. This pragma is enabled in the header file channel.h. If channel.h is included in the program this pragma will remain active until specifically disabled.
inline_ops	il	Compiles certain operations on long operands (signed or unsigned) on 16-bit targets as in-line operation rather than as calls to the compiler library. Operations affected are: ~ (bitwise complement), +, –, & (bitwise AND), | (bitwise OR), ^(bitwise exclusive OR), <<,>>, <, <=, ==, !=, >=, and >. Default is on.
printf_checking	pc	Checks that arguments passed to a function conform to the format used by printf. Default is off. This pragma is normally used to check formal arguments which are to be passed directly as format strings to printf. For each function within the scope of the pragma the last formal parameter is read as a format string and subsequent variable arguments are checked for correct type, according to the formatting rules of printf. This pragma is enabled in stdio.h for the declaration of printf and related functions, and subsequently disabled.
scanf_checking	sf	Checks that arguments passed to a function conform to the format accepted by scanf. Default is off. Otherwise this pragma has the same effect printf_checking. This pragma is enabled in stdio.h for the declaration of scanf and related functions, and subsequently disabled.
stack_checking	sc	Checks for stack overflow at the start of each function. Default is off.
warn_deprecated	wd	Warns of parameterless function declarations. Default is on.
warn_implicit	wi	Warns of undeclared functions. Default is on.

Table 1.6 Parameters to IMS_on and IMS_off

Pragma IMS_place_at_workspace_offset

This pragma directs the compiler to allocate a local (automatic) variable at *offset* words from the base of the current function's local workspace. The compiler will ensure that at least *offset* words are allocated, and that no other variables in scope at the same time as *local.variable* are allocated in the same place in workspace.

Syntax: #pragma IMS_place_at_workspace_offset (*local.variable, offset*)

This pragma is useful when using transputer instructions which require that values are placed at fixed workspace offsets, or which may overwrite values in workspace. For example, the *disc* instruction writes to workspace location '0', so to ensure that no other local variables will be allocated in this location, a dummy variable can be declared and placed at this location, as in the following example:

```
{
  int dummy;
  #pragma IMS_place_at_workspace_offset (dummy, 0)

  __asm {
    ... set up the alt
    ... disabling code
    disc;
    ... more disabling code
    }
 }
```

The named local variable must have been defined before the pragma is encountered.

A warning is generated and the pragma is ignored if:

- the named variable has not been defined before the pragma is encountered;
- the named variable is not local (automatic);
- the workspace offset supplied is negative.

Pragma `IMS_translate`

The compiler replaces all references to *name* (e.g. an external routine) by “*newname*”.

Syntax: `#pragma IMS_translate (`*name*`,"`*newname*`")`

“*newname*” is a C string which can contain alphanumeric characters; the underscore (‘_’), percent (‘%’), or full stop (‘.’) characters.

Pragma `IMS_trap_handler`

This pragma indicates to the compiler that the named function is to be used as a trap handler.

Syntax: `#pragma IMS_trap_handler (`*function-name*`)`

The named function must have been declared before the pragma is encountered.

A warning is generated and the pragma is ignored if,

- the named function has been defined before the pragma is encountered;
- its parameter is not a function identifier;
- it is given no parameters or more than one parameter;
- the target processor is not a T450 or ST20.

Where support is provided by the toolset for trap handling, details of how to use the trap handling facilities are given in the accompanying ‘*User Guide*’.

1.5.12 #undef

Syntax: `#undef` *identifier*

This directive causes the current definition of *identifier* (as defined using the `#define` directive) to be deleted.

1.6 Compiler predefinitions

Certain macros which identify global information, and some function names, are automatically recognized by the compiler. Generally, these items can be referenced directly in C programs and do not need to be declared.

Note: Predefined variables `_lsb` and `_params` (see section 1.6.2) should be declared to avoid spurious warning messages being generated by the compiler.

1.6.1 Macro names

All predefined macro names defined by the ANSI standard are present; they are:

`__DATE__`	—	The date of compilation of the source file.
`__FILE__`	—	Name of the current source file.
`__LINE__`	—	Line number of the current line of source.
`__STDC__`	—	The decimal constant '`1`' showing the implementation conforms to ANSI C.
`__TIME__`	—	The time of compilation of the source file.

The following SGS-THOMSON macro names are also defined:

`__CC_NORCROFT`	—	Derived from the Norcroft C compiler.
`_ICC`	—	SGS-THOMSON C compiler.
`_PTYPE`	—	Processor type.
`_ERRORMODE`	—	Execution error mode.
`__SIGNED_CHAR__`	—	Signedness of the plain `char` type, defined if the `icc` '`FC`' command line option is used.
`__FULL__DEBUG__`	—	Full debugging data, defined if the `icc` '`G`' command line option is used.
`__ICC_VERSION_STRING__`	—	Compiler version string.
`__ICC_VERSION_NUMBER__`	—	Compiler version number.

Details of the macros and the values they can take can be found in chapter 4 of the '*ANSI C Language and Libraries Reference Manual*'.

1.6.2 Other predefines

Two further names `_lsb` and `_params` are predefined by the compiler. They can be used in expressions in the same way as C variables. Both represent addresses which may be manipulated in low level programming and *must* be declared as follows:

```
extern volatile const void *_lsb;

extern volatile const void *_params;
```

_lsb is a pointer to the base of the compiled file's data area.

_params is a pointer to the base of the the current function's parameter block. It can be used to obtain low level information about a function's runtime code.

The following example illustrates how _params can be used to determine a function's return address, global static pointer, and workspace pointer.

```
void p()
{
 extern volatile const void *_params;
 typedef struct paramblock

    { void *return_address;
      void *gsb;
     int regparaml, regparam2;
    }
     paramblock;

 paramblock *pp = (paramblock *)_params;

 /* Return address is: pp—>return_address
    global static base sb is: pp—>gsb
    caller Wptr is: (void *) (pp + 1)   */
}
```

1.7 Transputer inline code

The ANSI C toolset provides different levels of support for inlining transputer instructions:

- A special keyword __asm can be used to insert sequences of transputer instructions into C programs. The __asm statement and how to use it is described in chapter 4 of the '*ANSI C Language and Libraries Reference Manual*'.

- A number of functions are supplied which can be compiled inline as transputer instructions, provided the appropriate header files are included in the source code. The inputs and outputs of the instructions are treated as parameters to and results from the functions.

1.7.1 Inlined functions

Each of the supplied functions is designed to allow access to a transputer instruction which is not directly accessible from the C source level. **Note:** however, that the automatic inlining will only occur if the appropriate header file has been incorporated in the source code by using the #include directive. The header files contain prototypes for the routines. Table 1.7 lists the functions, the instructions they support and the header file which is required.

Function	Instruction supported	Header file
`BitCnt`	*bitcnt*	`misc.h`
`BitCntSum`	*bitcnt*	`misc.h`
`BitRevNBits`	*bitrevnbits*	`misc.h`
`BitRevWord`	*bitrevword*	`misc.h`
`BlockMove`	*move*	`misc.h`
`CrcByte`	*crcbyte*	`misc.h`
`CrcWord`	*crcword*	`misc.h`
`DirectChanIn`	*in*	`channel.h`
`DirectChanInChar`	*in*	`channel.h`
`DirectChanInInt`	*in*	`channel.h`
`DirectChanOut`	*out*	`channel.h`
`DirectChanOutChar`	*outbyte*	`channel.h`
`DirectChanOutInt`	*outword*	`channel.h`
`memcpy`	*move*	`string.h`
`Move2D`	*move2dall*	`misc.h`
`Move2DNonZero`	*move2dnonzero*	`misc.h`
`Move2DZero`	*move2dzero*	`misc.h`
`ProcGetPriority`	*ldpri*	`process.h`
`ProcReschedule`	–	`process.h`
`ProcTime`	*ldtimer*	`process.h`
`SemSignal` †	*signal*	`semaphor.h`
`SemWait` †	*wait*	`semaphor.h`
`strcpy`	–	`string.h`
† Supported by T450 and ST20 targets only.		

Table 1.7 Inlined functions

Note: the '`DirectChan`...' functions must not be used with software virtual channels: the chapter on 'Configuration' in the '*User Guide*' discusses this, for toolsets which support software virtual channels.

Descriptions of all the functions are provided in the '*ANSI C Language and Libraries Reference Manual*'.

1.8 Compiler diagnostics

This section lists diagnostic messages generated by `icc`. The section is introduced by descriptions of some standard terms which may be encountered in the message texts.

1.8.1 Message format

Diagnostic messages are displayed in the standard toolset format for error messages. Details of the standard can be found in appendix A.

1.8.2 Severities

Diagnostics are tagged with a severity level which indicates their effect on the compilation. Severity levels are the same as those used in the toolset standard but have slightly different meanings, which are described below.

Information messages provide the user with information about the functioning or performance of the tool. They do not indicate an error and no user action is required in response.

Warning severity diagnostics are generated whenever legal, but unorthodox programming styles are detected. Compilation is unaffected and object code is generated normally.

Error severity diagnostics are generated whenever the compiler detects a programming error from which it can recover. Compilation continues, but may abort if more errors are detected subsequently. No object code is generated.

Serious severity diagnostics are generated when programming errors are detected from which the compiler cannot recover. Compilation continues but code has been lost. No object code is generated.

Fatal errors indicate internal inconsistencies in the software and cause immediate termination of the operation with no output. Fatal errors are unlikely to occur but if they do the fact should be reported to your local SGS-THOMSON distributor or field applications engineer.

Error, Serious, and *Fatal* diagnostic messages return error codes for handling by system MAKE programs and batch files.

1.8.3 Standard terms

This section explains some of the standard terms and notation used in compiler error messages.

abstract declarator

When using explicit casts or when passing an argument to `sizeof()`, a data type must be specified. This can be done by declaring an object of the correct type without specifying the name of the object. Declarations of this type are called abstract declarations, because they apply to no known object.

Examples of abstract declarations are:

```
(int) a = b;       /* 'int' is the abstract
                    declarator */

sizeof(int [3]);   /* 'int [3]' is the abstract
                    declarator */
```

char

Stands for a single ASCII character.

context

A context, e.g. ‘case expression’.

count

A number.

deprecated declaration

This means that a function declaration is incomplete. Declarations should specify the type of the function and the type of each formal parameter. If there are no parameters then the function type `void` should be specified.

expression

Stands for a C expression.

filename

A file name.

function prototype

A function declaration which usually precedes the function definition. It declares the function's type and the types of its parameters.

identifier

A C identifier, for example, a variable or function name.

initializer

An initial value which is assigned to an object at the time of its declaration.

instruction

A transputer instruction, or a pseudo-instruction as accepted by the `__asm` construct.

number

A number.

op

An operator. Valid operators include: “++”, “—”, “–>”, “<=”, and the unary operators `&`, `*`, + and –.

store class

A C storage class. Valid classes are `static` or `extern`.

string

Any string of ASCII characters.

struct/union

`struct` or `union`.

symbol

A C token.

type

A type identifier.

void context

This can occur at any point in a program where a value is not expected, for example, calling a function without using the returned number.

1.8.4 Information messages

These messages are prefixed by the word '**Information –**' ; they do not signal any required response from the user.

Function *identifier* has been marked as side effect free

The compiler has checked that the named function is side effect free and has marked it as such, from this point on in the compilation. Marking the function as side effect free may improve the generated code.

1.8.5 Warning diagnostics

These messages are prefixed by the word '**Warning –**' and indicate that unexpected results may occur.

#define macro *identifier* defined but not used

The named macro has been defined, but not referenced in the rest of the program. This message is only generated if specifically enabled by the '`FM`' compiler option.

'&' unnecessary for function or array *identifier*

A pointer to a function or array is implied by use of the name alone; the '`&`' operator is not required.

***identifier* already has a descriptor defined – pragma ignored**

The pragma `IMS_descriptor` has already been applied to *identifier*; more than one application is invalid.

***identifier* has been called – pragma ignored**

The pragma must be applied to *identifier* before the latter has been called.

***identifier* has been defined – pragma ignored**

The pragma must be applied to *identifier* before the latter is defined.

***identifier* has not been declared – pragma ignored**

The pragma must be applied to *identifier* after the latter has been declared.

***identifier* is not a function – pragma ignored**

The argument to the pragma must be a function name.

identifier **is not a local variable - pragma ignored**

The pragma `IMS_place_at_workspace_offset` has been applied to *identifier*, but the latter is not a local variable.

identifier **is not a variable - pragma ignored**

The pragma `IMS_place_at_workspace_offset` has been applied to *identifier*, but the latter is not a variable.

identifier **is not externally visible – pragma ignored**

The first argument to the `IMS_descriptor` pragma must be the name of an externally visible function.

identifier **may be used before being set**

The compiler has detected a use of a variable which may not have been initialized.

identifier **multiply translated, this translation ignored**

The `IMS_translate` pragma has been applied to *identifier* more than once.

number **treated as** *number* **ul in 32-bit implementation**

No type was specified for the number. The compiler assumes `unsigned long` if no type was specified.

op: **cast between function pointer and non-function object**

The operation is performed upon two arguments, one of which is a function, and the other an object.

A very suspect way of writing through a pointer has been detected in *function* **which is marked as side effect free**

The named function is marked side effect free and has some code in it to write through a pointer either in an unportable manner or in a way that is typically considered bad programming practice, e.g. *(int *)23 = ...; it is up to the user to carefully check that the assignment is conformant with the definition of side effect free.

Actual type *type* **mismatches format '%***char***'**

The type of an argument to `printf` or `scanf` does not match that implied by the control string.

ANSI '*char char char***' trigraph for '***char***' found – was this intended?**

The specified three character sequence was found in the source program. This has been treated as an ANSI trigraph and substituted for the character shown.

Argument and old-style parameter mismatch: *expression*

There is an old (non-prototype) style function definition in scope, and the type of an argument (after default argument promotion has taken place) does not agree with the type of the corresponding formal parameter.

Be sure that assignment through pointer in *identifier* is side effect free

The named function is marked side effect free and assigns through a pointer; it is up to the user to carefully check that the assignment is conformant with the definition of side effect free.

Be sure that functions pointed to in *identifier* are side effect free

The named function is marked side effect free and calls functions through pointers to them; it is up to the user to carefully check that the functions which may be called in this way are themselves side effect free.

Be sure that the assembler in *identifier* is side effect free

The named function is marked side effect free and uses assembly language inserts; it is up to the user to carefully check that assembler is conformant with the definition of side effect free.

Both an asterisk and an integer have been given for a field width

Only one of '*' or a decimal integer is valid for a `printf` field width. This message is only generated if `IMS_on`(pc) is active. The header file `stdio.h` includes this pragma.

Both an asterisk and an integer have been given for a precision

Only one of '*' or a decimal integer is valid for a `printf` precision. This message is only generated if `IMS_on`(pc) is active. The header file `stdio.h` includes this pragma.

Cannot delete temporary file *filename*

Host file system error.

Cannot generate stack check for *identifier* (pragma nolink applied)

A stack check requires a static link, and the function *identifier* has been specified not to receive a static link (using `IMS_nolink`). `icc` compiles the function with the stack check omitted.

Character sequence /* inside comment

The start-of-comment character sequence was detected within a comment. Nested comments are not legal in C. Check that the previous comment was terminated correctly.

Code relies on non–portable form of constant expression in initialiser

Objects that have static storage duration should only be initialized by constant expressions, which does not include casts of pointer expressions to integral types. However, `icc` does support this extension as long as the integral type has the same size as a pointer type.

Dangling 'else' indicates possible error

Within nested `if ... else` constructs, there is some ambiguity as to which 'if' relates to which 'else'.

Deprecated declaration '*identifier* ( )' – give arg types

In the prototype declaration of the named function, the arguments' types were not specified.

Division by zero: *op*

Division, or remainder, by zero, will cause overflow.

Empty body in an else-statement

A null statement (i.e. just a semicolon) has been found as the body of an `else` statement. The `else` statement has no effect, therefore this may indicate a mistake.

Empty body in an if-statement

A null statement (i.e. just a semicolon) has been found as the body of an `if` statement; this may indicate a mistake.

Enumeration constant *identifier* **declared but not used**

The named enumeration constant was declared but not used within its scope.

Expected ')'; perhaps you tried to give too many names – pragma ignored

A ')' was expected but not found in a pragma; it may be that too many parameters have been given.

Expected integer as argument – pragma ignored

An integer argument was expected but not found in a pragma.

Expected string as argument – pragma ignored

The argument to the `IMS_linkage` pragma must be a string literal.

Expected string as fifth argument – pragma ignored

The fifth argument to the `IMS_descriptor` pragma must be a string literal.

Expected string as second argument – pragma ignored

The second argument to the `IMS_translate` pragma must be a string literal.

Extern 'main' needs to be 'int' function

In a declaration of `main`( ), the function should always be declared as type `int`.

Extern *identifier* **not declared in header**

All objects must be declared before use. This message is only generated if specifically enabled by the '`FM`' compiler option.

File *filename* **is empty**

A source file, or a file which is `#included` contains no characters.

Floating point constant overflow: *op*

Floating point overflow occurred during addition, subtraction, multiplication or division of two constants.

Floating to integral conversion failed

Conversion (casting) from a floating point type to an integral type (such as `int`) failed.

Formal parameter *identifier* may not be placed at a fixed workspace offset

The pragma `IMS_place_at_workspace_offset` has been applied to a formal parameter, but this is not allowed in this implementation.

Formal parameter *identifier* not declared – 'int' assumed

A formal parameter has been listed in the parameter list of the function definition, but there is no entry for it in the declaration list; it is therefore assumed to be of type `int`.

Format requires *count* parameter(s), but *count1* given

A call to `printf` or `scanf` was made with the incorrect number of arguments. The control string indicated that *count* arguments are needed, but *count1* were provided. This warning is only generated for `printf` if `IMS_on(pc)` is active, and for `scanf` if `IMS_on(sf)` is active. The header file `stdio.h` includes these pragmas.

Function *identifier* declared but not used

The function was declared at block scope but was not called within the block.

Function *identifier* marked as an interrupt handler with no static link - this has undefined effects

The named function has both the `IMS_interrupt_handler` and the `IMS_nolink` pragmas applied to it. This does not make sense because the code which sets up an interrupt handler always makes sure that a static link is available.

Function *identifier* marked as a trap handler with no static link - this has undefined effects

The named function has both the `IMS_trap_handler` and the `IMS_nolink` pragmas applied to it. This does not make sense because the code which sets up a trap handler always makes sure that a static link is available.

Global optimisation suppressed for *identifier* as it contains assembly code

The user has attempted to globally–optimize a function which contains an assembler insert: the compiler automatically turns the global optimizer off. (This applied only to the named function: the global optimizer will be turned on again for subsequent functions.)

Illegal format conversion '%*string*'

The character sequence '%*string*' is not a legitimate conversion specification for `printf` or `scanf`. This warning is only generated for `printf` if `IMS_on(pc)` is active, and for `scanf` if `IMS_on(sf)` is active. The header file `stdio.h` includes these pragmas.

Illegal language type *string*; replaced by *string*

The language type given for the interface descriptor–string is not a valid one, and has been overridden by a known type.

Implicit cast (to *type*) overflow

Overflow occurred when casting an expression.

Implicit narrowing cast: *op*

The result of an operation or expression performed at higher precision is immediately, and implicitly, cast to lower precision, thus losing the extra precision: if the extra precision is not required, the operation ought to be performed at the lower precision.

If the narrowing cast is really required, the warning may be suppressed by writing the cast explicitly or by specifying the '`WN`' command line option.

Implicit return in non-void function *identifier*

The function does not contain a `return` statement, even though it is defined to return a value.

IMS_interrupt_handler has no effect on this target

The `IMS_interrupt_handler` pragma has been used when compiling for a target processor which does not support interrupt handlers. The compiler will ignore the use of this pragma.

IMS_trap_handler has no effect on this target

The `IMS_trap_handler` pragma has been used when compiling for a target processor which does not support trap handlers. The compiler will ignore the use of this pragma.

Incomplete format string

The control string for use with `printf` or `scanf` is incomplete. This warning is only generated for `printf` if `IMS_on(pc)` is active, and for `scanf` if `IMS_on(sf)` is active. The header file `stdio.h` includes these pragmas.

'int *identifier* ()' assumed – 'void' intended?

A function was defined without specifying its return type. The compiler assumes a return type of `int` if no type is specified.

Integer too large to be represented – pragma ignored

An integer parameter to a pragma has been given with a value too large to be able to be dealt with by the compiler.

Inventing 'extern int *identifier* ();'

No declaration exists for the function; it will be defined by default as `extern int`, with no information about its parameters.

'*j*' to immediately following label *identifier* will be removed

In an assembler insert, there is a *j* (jump) instruction to an immediately following label. This is effectively a no–op, and is removed.

If the user really requires a *j 0* instruction, for breakpointing or descheduling purposes, he should write *j 0* in the assembler insert.

Label *identifier* was defined but not used

The named label was set, but not used.

Linkage already set – pragma ignored

The `IMS_linkage` parameter has been specified more than once.

Lower precision in wider context: *op*

The result of an operation performed at lower precision is immediately cast to a higher precision; it may be that the user was expecting the operation to be performed at the higher precision.

This warning may be disabled by using the command line option '`WN`' or by inserting a cast to the lower precision. Thus indicating that the operation is really expected to be performed at the lower precision. For example, in the following code fragment:

```
void f(void)                void f(void)
{                           {
  int i;         becomes:     int i;
  long int l;                 long int l;
  l = i + i;                  l = (int)(i + i);
}                           }
```

Missing comma in pragma argument list – pragma ignored

Multiple arguments to a pragma must be separated by commas.

Missing macro argument

One of the parameter slots in a macro invocation is empty.

Negative value given for vectorspace – pragma ignored

Vectorspace values in the `IMS_descriptor` pragma must be ≥ 0.

Negative value given for workspace – pragma ignored

Workspace values in the `IMS_descriptor` pragma must be ≥ 0.

Negative value given for workspace offset – pragma ignored

Workspace values in the `IMS_place_at_workspace_offset` pragma must be ≥ 0.

No parameter(s) given – pragma ignored

One or more parameters were expected to be given to the pragma, but none were found.

No pragma name given in pragma directive – was this intended?

The compiler has detected a pragma directive which does not have a name. This is not illegal, however, it has no effect.

No side effect in void context: *identifier*

The value which has been returned by an expression is not being used e.g.

```
int a;
a;
```

Non-portable – not 1 char in '. . .'

The characters enclosed by single quotes represent more than one character. The compiler will read the first character only, for example, 'AB' will be read as 'A'.

Non-positive values for patch size are meaningless – pragma ignored

Patch size values must be > 0.

Non-value return in non-void function

A function which should return a value has terminated without using a return statement or with a return statement that has no arguments. The value received from the function by the calling routine is undefined.

Odd unsigned comparison with 0 : *op*

a ≥ comparison of an unsigned integer with zero, or a ≤ comparison of zero with an unsigned integer, is always true.

Omitting trailing '\0' for char [*count*]
Omitting trailing '\0' for wchar_t [*count*]

The char array is fully occupied by characters and there is no room to append the string terminator (\0). *count* is the full length of the character array.

Only one asterisk as field width has meaning

More than one '*' has been given as a field width in a `printf`. This message is only generated if `IMS_on`(pc) is active. The header file `stdio.h` includes this pragma.

Only one asterisk as precision has meaning

More than one '*' has been given as a precision in a `printf`. This message is only generated if `IMS_on`(pc) is active. The header file `stdio.h` includes this pragma.

Option 'G' conflicts with option 'O*number*' – 'O*number*' ignored

Both debugging 'G' and optimization 'O*number*' options have been specified on the command line. The compiler cannot produce debugging information for optimized code and has therefore disregarded the optimization option.

Possible error: >= *number* lines of macro arguments

There are a surprisingly large number of lines of arguments to a macro; this may indicate a syntax error.

Pragma *identifier* has no effect in preprocess only mode

The named pragma has been recognized but its effect is not entirely local to the preprocessor, and hence in preprocessing only mode the pragma directive cannot have the behavior defined for it.

Repeated definition of #define macro *identifier*; previous on command line

The named macro has been defined more than once. The first definition was on the command line. The definitions are identical.

Repeated definition of #define macro *identifier*; previous on line *number* of *filename*

The named macro has been defined more than once. The first definition was on line *number* of the file *filename*. The definitions are identical.

Shift of *type* by *count* undefined in ANSI C

A shift of more than the number of bits in *type*, or less than zero was requested, and this is undefined in ANSI C.

Signed constant overflow: *op*

Overflow occurred when performing *op* upon signed, constant operands.

Spurious {} around scalar initialiser

A scalar can take only one initializer, so there is no need to use braces as are required with aggregate types such as arrays.

Static *identifier* declared but not used

The named static object was declared but not used.

'struct *identifier*' has no named member

A structure has been declared without any named members.

The combination of a precision and the conversion specifier *char* is undefined

A precision, introduced by a period '`.`', in a `printf` format string only has a meaning with certain conversion specifiers; an undefined combination has been detected. This message is only generated if `IMS_on`(pc) is active. The header file `stdio.h` includes this pragma.

The combination of the flag *char* and the conversion specifier *char* is undefined

The flags '`#`' and '`0`' in a `printf` format string have a meaning only with certain conversion specifiers; an undefined combination has been detected. This message is only generated if `IMS_on`(pc) is active. The header file `stdio.h` includes this pragma.

The combination of the short/long indicator *char* and the conversion specifier *char* is undefined

The short/long indicators 'h', 'l' and 'L' in a printf/scanf format string have a meaning only with certain conversion specifiers; an undefined combination has been detected. This message is only generated for printf if IMS_on(pc) is active, and for scanf if IMS_on(sf) is active. The header file stdio.h includes these pragmas.

Typedef *identifier* declared but not used

The named identifier has been declared, but is not used in its scope.

Undefined declaration of 'enum' type

The use of an enumeration tag has been detected before the content of that enumeration has been defined. Declarations of enumerations that have not yet had their content defined are not assigned any meaning by the ANSI C Standard and as such lead to un-portable code.

Undefined macro *string* in #if – treated as 0
Undefined macro *string* in #elif – treated as 0

This error occurs when enumeration constants, keywords, etc. appear after the preprocessor #if or #elif directives. For example, if 'ab' and 'cd' are enumeration constants, the directive #if ab == cd would generate this error.

'union *identifier*' has no named member

A union has been declared without any named members.

Unnamed bit-field ignored during initialisation

All unnamed structure or union members are ignored during initialization.

Unrecognised #pragma (no '(')

The arguments to a pragma are not correctly enclosed in parentheses.

Unrecognised #pragma (no ')')

The arguments to a pragma are not correctly enclosed in parentheses.

Unrecognised #pragma *identifier*

identifier is not a pragma recognized by this compiler.

Unrecognised pragma parameter *string* – pragma ignored

The pragma can take a limited range of parameters and the one given was not from this range.

Unsigned constant overflow: *op*

Overflow occurred when performing *op* upon unsigned, constant operands.

Unused earlier static declaration of *identifier*

There is a forward declaration of *identifier* which is not necessary as the definition of *identifier* appears before *identifier* is referenced.

Use of *op* in condition context

Generated when the operators '=' (assignment) or '~' (bit-not) are used in a condition statement.

This message is given for use of the assignment operator in condition context, e.g.

```
if (a = b)
```

as this is often due to mistyping the equality operator, i.e. the desired code is:

```
if (a == b)
```

If you really wish to perform the assignment in condition context, the warning message may be suppressed using the form:

```
if ((a = b) != 0)
```

Value of *identifier* is undefined as it has been assigned since the last sequence point

The compiler has detected that a variable has been assigned to and then subsequently read. However, the assignment and the read are not separated by a sequence point, and therefore the order in which they occur is undefined.

Variable *identifier* declared but not used

The variable was declared, but not used anywhere in its scope.

Variable *identifier* is assigned more than once between sequence points

If a variable is assigned more than once between sequence points, the order in which the assignments occur is undefined. In addition if the assignments give different values to the variable, then the subsequent value of the variable is undefined.

Workspace clash between *identifier1* and *identifier2* PLACEd at workspace *number*

Both identifiers are variables which have been placed at the same workspace offset using the pragma `IMS_place_at_workspace_offset`. However, the variables are both in use at the same time and so may conflict with each other. The compiler will still allocate both variables at the specified offset.

Wrong number of parameters to *identifier*

A function declared without a prototype was called with the wrong number of arguments. (An error is given if a function declared with a prototype is called with the wrong number of arguments.)

1.8.6 Recoverable errors

These messages are prefixed by the word '**Error –**'.

## first or last token in #define body

The `##` preprocessor operator must be both preceded and succeeded by a preprocessor token.

‘,’ (not ‘;’) separates formal parameters

A semicolon has been used to separate the formal parameters in a function definition (as in Pascal) instead of a comma.

<int> *op* **<pointer> treated as <int>** *op* **(int) <pointer>**

The expression involving a integer and a pointer will result in the pointer being converted (cast) to an integer.

identifier **marked as side effect free assigns to a global variable**

An assignment to a global variable is a side effect.

identifier **marked as side effect free assigns to static**

An assignment to a static variable, other than an initialization, is a side effect.

identifier **marked as side effect free calls** *identifier* **which is not side effect free**

The call of a function which is not side effect free is a side effect.

identifier **marked as side effect free uses volatile variable**

The read of or write to a volatile variable is a side effect.

instruction **may not have a size specified**

An `__asm` pseudo-instruction may not be explicitly sized.

op **: cast between function pointer and non-function object**

The operation is performed upon two arguments, one of which is a function, and the other an object.

op **: cast to non-equal** *type* **illegal**

A structure or union has been cast into a structure or union of a different type. The cast is illegal and will be ignored.

op **: illegal cast to** *type*

An illegal cast has been attempted. The cast will be ignored.

op **: implicit cast of** *type* **to ‘int’**

A non-integer object has been used where an `int` was expected, for example, attempting to use a `double` as an argument to a switch statement (which requires an integer type).

op **: implicit cast of non-0 int to pointer**

Evaluation of the expression will result in the cast of an integer to a pointer.

op **: implicit cast of pointer to ‘int’**

Evaluation of the expression will result in the cast of the pointer to an integer.

op **: implicit cast of pointer to non-equal pointer**

Evaluation of the expression will result in the cast of one pointer type to another.

op **may not have whitespace in it**

Compound assignment operators such as '+=' must not contain whitespace.

<pointer> *operator* **<int> treated as (int) <pointer>** *operator* **<int>**

Evaluation of the expression will result in the cast of the pointer to an integer.

Ancient form of initialisation, use '='

A `}`, rather than =, was used to introduce an initializer, this is no longer legal C.

ANSI C does not support 'long float'

An object has been declared of type `long float`, this is illegal in ANSI C, which supports `float`, `double`, or `long double.`

Array of *type* **illegal — assuming pointer**

An array of functions or void objects has been declared. The compiler treats this as an array of pointers to functions or void objects.

Array [0] found

An empty array has been defined and will be set up instead as an array with one element.

Assignment to 'const' object *identifier*

The expression contains an assignment to a constant.

Assignment to *identifier* **which has 'const' qualified member**

An attempt has been made to assign to a constant. The named object is a structure or union which has a member, or recursively has a member of a contained structure or union, which is constant.

Attempt to do arithmetic on pointer to an incomplete type

Pointer arithmetic must be done using pointers to object types. This includes the fact that the pointer operand of array subscription must be the pointer to an object, not incomplete, type.

Attempt to take address of *identifier* **which has 'register' storage class**

It is illegal to take the address of a variable with '`register`' storage class.

Cannot use pointer to type *type* **in arithmetic**

Pointer arithmetic must be done using pointers to object types.

'const' typedef *identifier* **has 'const' re-specified**

A typedef which is already qualified with `const`, has been qualified with `const`.

Comparison *op* **of pointer and int: literal 0 (for == and !=) is only legal case.**

The specified operator was used to compare an object of type `int` and one of a type `pointer`. The only legal comparison of this type is between a pointer and 0 using either == or `!=`.

Declaration with no effect

No name has been declared for the object. For example, specifying only the type of an object generates this error.

Differing pointer types: *op*

The specified operator was used with pointers of different types.

Differing redefinition of #define macro *identifier*: **predefined macro name**

The named macro was predefined by the compiler, and has been redefined to a different value.

Differing redefinition of #define macro *identifier*: **previous on command line**

The named macro was defined on the command line, and has been redefined to a different value.

Differing redefinition of #define macro *identifier*: **previous on line** *number* **of** *filename*

The named macro has been defined more than once. The definitions are not identical.

Differing struct/union types: op

The specified operator was used with `structs`/`unions` of different types.

Digit 8 or 9 found in octal number

8 and 9 are meaningless in an octal number.

Duplicate macro formal parameter: *identifier*

The function macro has two formal parameters with the same name.

Duplicate member *identifier1* **of** *identifier2*

Two fields of structure or union *identifier2* have the name *identifier1*.

Ellipsis (...) cannot be only parameter

A function declared to take a variable number of parameters must have at least one known parameter.

Enumeration constant *identifier* **too large to represent as 'int' – 0 assumed**

The value of an enumeration constant has overflowed the range of `ints`.

Escaped newline must not immediately precede EOF – newline inserted

A non–empty source file must end in a newline which is not immediately preceded by backslash.

Extern *identifier* **mismatches top-level declaration**

An `extern` declaration of *identifier* within a function definition does not match an `extern` declaration of *identifier* at the top level.

Expected *symbol1* **or** *symbol2* **–inserted** *symbol1* **before** *symbol3*

symbol1 or *symbol2* was expected before *symbol3*, but neither was found. *symbol1* is suggested as the most appropriate choice and the compiler has changed the code accordingly.

Formal name missing in function definition

The name of a formal parameter has been omitted in a function definition.

Function *identifier* **may not be initialised – assuming function pointer**

Initializers cannot be used in function declarations or definitions.

Function prototype formal *identifier* **needs type or class – 'int' assumed**

At least one of type specifier, type qualifier (`const` or `volatile`) or the storage class `register` is needed when declaring a function prototype formal parameter. An `int` has been assumed.

Function return type *struct/union* **is incomplete — assuming pointer**

The structure or union type that the function returns has not yet been completed but must be before the function can be defined or called.

Function returning *type* **illegal — assuming pointer**

It is illegal for a function to return a function or an array.

Hex number cannot have exponent

A hex number ending in `e` may not be immediately followed by + or -; separate the number and the additive operator with white space. In terms of the ANSI C standard, a valid preprocessing number (*pp-number*) has been found, but it is not a valid constant.

Illegal bit-field type *type* **– 'int' assumed**

A bit-field can only have a type that is a qualified or unqualified version of one of `int`, `unsigned int`, or `signed int`. The compiler assumes an `int` instead.

Illegal format for command line macro definition *string*

The format must be *macro_name* or *macro_name=value*, where *macro_name* is a valid macro name (thus having the same syntax as an *identifier*). In the second form, the equals sign must NOT be separated from *macro_name* or *value* by any white space.

Illegal escape sequence '\*char*' – treated as '*char*'

The character following \ does not form part of a valid escape sequence. The compiler treats the sequence \char as *char*.

Illegal [] member: *identifier*

An open array may not be a member of a structure or union.

Junk at end of #*identifier* line

The text following the directive is invalid.

Linkage disagreement for *identifier* – treated as *store class*

The storage class of a previously defined `static` or `extern` object or function disagrees with the current declaration. The object will be treated as though it is in storage class *store class.*

Maximum object size exceeded; limit is *number*

An object has been declared which exceeds the maximum allowed for the target processor. The maximum size an object may be declared to be is *number* bytes.

Member *identifier* may not be function – assuming pointer

An attempt was made to declare a function as a structure or union member, which is invalid.

Missing newline before EOF – inserted

Non-empty source files must end with an un-escaped newline.

Missing type specification – 'int' assumed

A type specification is missing. The object will be assumed to be of type `int`.

Negative numbers and zero are not allowed in #line

ANSI C forbids negative numbers or zero in a #line directive.

No chars in character constant ''

No characters or character codes have been specified for the character constant. A NULL character is assumed.

No initializer list in braced initializer

There must be at least one entry in the initializer list of a braced initializer.

Number illegally followed by letter, underscore or period

A numerical constant may not be followed immediately by a letter, underscore or period.

Number missing in #line

There is no line number following the preprocessor `#line` directive.

Numbers greater than 32767 are not allowed in #line

ANSI C forbids numbers greater than 32767 in a #line directive.

Object *identifier* may not be function – assuming pointer

An attempt was made to declare a function where such a declaration is not valid, e.g. a block scope declaration with storage class `static`.

Omitted `<type>` **before formal declarator – 'int' assumed**

No type was specified; type int will be assumed.

Operand of # not macro formal parameter

The operand to the # preprocessor operator must be a formal parameter of the function macro containing it.

Overlarge escape '\*number1*' treated as '\number2'

An octal number in an escape sequence is too large to be represented in the target architecture.

Overlarge escape '\x*number1*' treated as '\x*number2*'

A hexadecimal number in an escape sequence is too large to be represented in the target architecture.

Parentheses (. . .) inserted around expression following *string*

Parentheses were expected after the specified string, for example, around a conditional expression such as an `if` statement.

Prototype and old-style parameters mixed

It is illegal to mix new (prototype) and old-style parameter declarations.

Return <expr> illegal for void function

A return statement with an expression was found within a `void` function. The return statement is ignored.

Size of 'void' required

'void' was used as an argument to `sizeof`.

Size of a [] array required

An array of unspecified size was used as an argument to `sizeof`.

Size of <function> required

A function name was used as an argument to `sizeof`.

Sizeof <bit-field> illegal

A bit-field was used as an argument to `sizeof`.

Size of '*struct/union identifier*' needed but not yet defined

The size of the structure/union cannot be determined. This error can occur when an undefined structure/union is used as an argument to the `sizeof` operator and when an undefined structure/union is used in the declaration of a variable.

Size qualifier must be a positive value.

The operand to `size` is the number of bytes that the instruction is to occupy, and therefore a negative or zero value is meaningless.

Small floating point value converted to 0.0

The number is too small to represent in floating point format, and has been rounded to 0.0.

Spurious #elif

The `#elif` directive could not be matched with a corresponding `if` directive.

Spurious #else

The `#else` directive could not be matched with a corresponding `if` directive.

Spurious #endif

The `#endif` directive could not be matched with a corresponding `if` directive.

Static function *identifier* not defined – treated as extern

A function was defined as `static` in the function prototype, but the compiler was unable to find the function definition. An `extern` function is assumed.

String initialiser longer than char [*count*]
String initialiser longer than wchar_t [*count*]

A character array has been initialized with more characters than the array can accommodate. Since the compiler adds a terminating NULL character to strings, string initializers should always contain one less element than the array.

'struct *identifier*' has no members

A structure definition must contain at least one member.

Struct member *identifier* may not be function – assuming pointer

A structure member was declared of function type; the compiler treats this as pointer to function type.

Translation unit contains no external declarations

A translation unit must contain at least one external declaration. A translation unit is a source file together with all the headers and source files included via the preprocessing directive `#include`, and less any source lines skipped by any of the conditional inclusion preprocessing directives.

Type specifier/qualifier or storage class needed –'int' assumed

At least one of type specifier, type qualifier (`const` or `volatile`) or storage class is needed when declaring an object or function.

Undeclared name, inventing 'extern int *identifier*'

An undeclared identifier was encountered and will be given the type `extern int`.

'union *identifier*' has no members

A union definition must contain at least one member.

Union member *identifier* **may not be function – assuming pointer**

A union member was declared of function type; the compiler treats this as pointer to function type.

Unprintable char *number* **found**

An unprintable character was found in the source text.

'volatile' typedef *identifier* **has 'volatile' re-specified**

A typedef which is already qualified with `volatile`, has been qualified with `volatile`.

Wrong number of parameters to *identifier*

A function was called with the wrong number of arguments.

1.8.7 Serious errors

These messages are prefixed by the word '**Serious –**'.

{} must have 1 element to initialise scalar

When initializing a scalar variable only one initializer should be specified within the enclosing braces.

#error encountered *string*

The `#error` directive was found.

#include file *filename* **wouldn't open**

The file *filename* could not be opened.

op **: cast to non-equal** *type* **illegal**

A structure or union has been cast into a structure or union of a different type.

op **: illegal cast of** *type* **to pointer**

A variable has been cast into a pointer type.

op **: illegal cast to** *type*

An illegal cast has been attempted.

context: **illegal use in pointer initialiser**

An invalid way of initializing a pointer has been attempted.

(. . .) must have exactly 3 dots

An ellipsis must consist of three dots.

'{' of function body expected – found *string*

The opening brace in the body of a function is missing.

'{' or <identifier> expected after *type*, but found *string*

The opening brace following a `struct`, `union` or `enum` is missing. *string* marks the position.

<asm-directive> expected but found a *string*

string indicates what was found in place of the expected `__asm` directive.

<command> expected but found a *string*

Statements such as `switch` or `if` should be followed by a command. *string* indicates where the command was expected.

<expression> expected but found *string*

string indicates where the expression was expected.

<identifier> expected but found *string* in 'enum' definition

The compiler was expecting to read an enumeration constant when it found *symbol*. This may be because there is a spurious comma at the end of a list of enumeration constants.

identifier **has pragma nolink specified, but accesses static data**

The specified function has been specified not to receive a static link (via `IMS_nolink`), but attempts to use static data. It is only possible to use static data when a static link is available.

string **is not a label**

The operands to the `ldlabeldiff` pseudo-instruction must be labels.

identifier1 **has pragma nolink specified, but accesses static** *identifier2*

Function *identifier1* has had the `IMS_nolink` pragma applied to it, which means it cannot access static data.

identifier1 **has pragma nolink specified, but addresses static** *identifier2*

Function *identifier1* has had the `IMS_nolink` pragma applied to it, which means it cannot address static data.

instruction **not followed by label**

A jump, conditional jump or call `__asm` instruction must have a constant or label operand.

store class **variables may not be initialised**

Some types of C variables, such as those declared as `extern`, cannot be initialized.

A read error has occurred

An I/O error occurred during the reading of a file.

Array size *count* illegal – 1 assumed

Arrays cannot be larger than 0xffffff on a 32-bit target, or 65535 on a 16-bit target.

Attempt to apply a non-function

A name not declared as a function has been used in a context where a function should be.

Attempt to include *'struct/union identifier' object/member identifier* within itself

A structure or union declaration may not contain a field of the structure or union type, or a field which references another field.

Attempt to take the address of a bit-field

Elements of type bit-field in C structures cannot be addressed.

Bit-field size *number* illegal – 1 assumed

Bit-field sizes greater than 32 for 32-bit targets, or 16 for 16-bit targets are illegal. Negative sizes are also illegal.

'break' not in loop or switch

A break statement was encountered outside the scope of a loop or switch statement. A break at this point is illegal.

Cannot address built–in variable *identifier*

identifier is a built–in name, such as `_lsb` or `_params`, which cannot be addressed.

Cannot call *identifier* (it requires a static link)

An attempt has been made to call the specified function which requires a static link, from a function which has been specified not to receive a static link (via `IMS_nolink`).

Cannot do indirect call (it requires a static link)

An attempt has been made to call a function from a function which has been specified not to receive a static link (via `IMS_nolink`). All calls through function pointers are assumed to require a static link.

Cannot read indirect file *filename*

The file either does not exist, or does not have read permission.

Cannot write to built–in variable *identifier*

identifier is a built–in name, such as `_lsb` or `_params`, which cannot be assigned to.

'case' not in switch

A case label has been encountered outside the body of a switch statement. A case labelled statement at this point is illegal.

Char and wide (L"...") strings do not concatenate

A char string and a wide char string appear adjacently in the source text. Normally, adjacent strings in the source text are concatenated; however, this is not possible here, as they have different types.

'continue' not in loop

A continue statement has been encountered outside the body of a loop. A continue statement at this point is illegal.

'default' not in switch

A default prefix has been encountered outside the body of a switch statement. A default label at this point is illegal.

Digit required after exponent marker

Exponents of floating point numbers must be followed by a numeric character. The numeric character may be preceded by '+' or '–'.

Duplicate 'default' label

The default label has already been specified for the switch construct.

Duplicate definition of *identifier*

The named identifier has already been defined.

Duplicate definition of *struct/union* **tag** *identifier*

The named structure or union identifier has already been used.

Duplicate definition of label *identifier*

The specified identifier has already been used.

Duplicate type specification of formal parameter *identifier*

The specified parameter has been listed more than once in the function's formal parameter list.

Duplicated case constant: *constant*

The constant has been specified more than once in the same switch statement.

EOF in comment

The end-of-file was detected inside a comment.

Error reading feedback file

A file-read error occurred when trying to read the feedback file.

Expected *symbol*

symbol was expected.

Expected *symbol1* **– inserted before** *symbol2*

symbol1 was expected before *symbol2* and the compiler has changed the code accordingly. For example, in the code "`if (TRUE printf();`" the compiler would expect to find ')' before '`printf`'.

Expected *symbol1* **or** *symbol2*

Either *symbol1* or *symbol2* was expected.

Expected <identifier> after *op* **but found** *string*

The specified operator must be followed by an identifier. This error may occur after the structure member operator '`.`' and the structure pointer operator '–>'.

Expecting <declarator> or <type>, but found *string*

An identifier or type was expected at *string*. For example, the declaration '`typedef int *[3] test;`' generates this error.

Feedback file has incorrect format

The compiler cannot read the profiling feedback file because it does not have the correct format - check it has been correctly specified on the command line.

Feedback file is out of step with source

File position information in the profiling feedback file does not match the source file being compiled; probably because the source file has been altered since the feedback file was created. Also, check the feedback file has been correctly specified on the compiler command line.

'goto' not followed by label

The text following a goto statement does not represent a label.

Hex digit needed after 0x or 0X

The hexadecimal specifier `0x` must be followed by a valid hexadecimal digit. The compiler assumes a zero digit.

Identifier *identifier* **found in <abstract declarator>**

An identifier should not be used in an abstract declarator. This error is generated, for example, if `sizeof(int *test[3]);` is used instead of the correct form `sizeof(int *[3]);`.

Illegal context for character (*number* **= '***char***')**
Illegal context for character (hex code *number***)**

The character is in the source character set, but is only allowed in a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token. (An example of the latter is the replacement list in '`#define DOLLAR $`' if the macro `DOLLAR` is never used).

Illegal in *context*: *text*

Illegal expressions such as those involving division by zero generate this error.

Illegal in *expression*: **non constant** *identifier*

A constant is required in certain expressions, for example after a `case` prefix.

Illegal indirection on (void *): '*'

An attempt has been made to take the contents of the object pointed to by a pointer to void.

Illegal in lvalue: array *identifier*

A whole array is not a modifiable lvalue.

Illegal in lvalue: 'enum' constant *identifier*

Enumeration constants cannot be used as lvalues in an expression.

Illegal in lvalue: function *identifier*

Function names do not represent objects and so cannot be used as lvalues.

Illegal in the context of an lvalue: *op*

The operator *op* cannot appear in lvalue context.

Illegal type for operand: *op*
Illegal types for operands: *op*

The operator has been used with an invalid type. For example, it is illegal to use the structure member operator '`.`' with a variable of type `int`.

Illegal 'void' member/object: *identifier*

An object or member of a structure or union cannot be declared as being of type `void`.

Incomplete tentative definition of *identifier*

The declaration of *identifier* has gone out of scope before the declaration has been completed.

Invalid object file name

The argument to the command line option '`o`' is not a valid filename.

Invalid source file name

The source filename specified on the command line is not a valid file name.

Invalid use of command line option (*string***)**

string is not a recognized command line option, or it is being used incorrectly, e.g. it requires a parameter and none has been given.

I/O error writing *filename*

An error occurred when writing to the named file.

Junk after #if <expression>
Junk after #elif <expression>

The `#if` and `#elif` directive must be terminated by a newline character.

Junk after #include *filename*

The `#include` directive must be terminated by a newline character.

Label *identifier* **has not been set**

A label has been referenced but not set. This message will be generated if, for example, `goto` is used with an undefined label.

Misplaced 'else'

An `else` statement was found where it was not expected.

Misplaced '{' at top level – skipping block

An opening brace was found at the top level of a program when it was not expected, for example when not used as part of a function or structure definition.

Misplaced preprocessor character '*char***'**

A preprocessor directive character (# or \) was found where it was not expected.

Missing #endif at EOF

An `#endif` directive is missing. This error will not be generated until the last of the currently open files is about to be closed (ANSI standard does not require `#if` and `#else` statements to match in included files).

Missing '*char***' in preprocessor command line**

A 'quote' character is missing from a preprocessor directive. The missing character could be `'`, `<`, `>`, or `"`.

Missing ')' after *identifier* **(. . . on line** *number*

A closing parenthesis is missing from the macro invoked on line *number*.

Missing ',' or ')' after #define *identifier* **(. . .**

The list of parameters in a macro definition is either incomplete or has not been correctly terminated by a closing parenthesis.

Missing '<' or ' " ' after #include

The opening 'quote' character which introduces the filename is missing.

Missing hex digit(s) after \x

The hexadecimal introducer sequence `\x` was found, but no hexadecimal digit was specified. The compiler assumes that the letter `x` was intended.

Missing identifier after #define

The definition is empty. `#define` must be followed by an identifier.

Missing identifier after #ifdef

`#ifdef` must be followed by an identifier.

Missing identifier after `#ifndef`

`#ifndef` must be followed by an identifier

Missing identifier after #undef

`#undef` must be followed by an identifier.

Missing parameter name in #define ***identifier*** **(. . .**

A parameter is missing from the specified macro definition. This error would be generated by a definition of the form `#define test(arg,)`.

Multiple source files specified

Only one source file may be specified on the command line.

Newline or end of file within character constant

A newline or end-of-file was encountered within a character constant.

Newline or end of file within string literal

A newline or end-of-file was encountered within a string literal.

No ')' after #elif defined(. . .
No ')' after #if defined(. . .

The closing parenthesis is missing from the directive.

No identifier after #elif defined
No identifier after #if defined

`#elif defined` and `#if defined` must be followed by an identifier.

Non-formal ***identifier*** **in parameter-type-specifier**

The parameter *identifier* was included in the declarator list of a function, but not in the parameter list. For example, a definition such as `int foo() int x; {}` would generate this error.

Non-static address ***identifier*** **in pointer initialiser**

Static pointers and fields of `structs` or `unions` cannot be initialized with the address of an object of type `auto`.

Number ***number*** **too large for 32-bit implementation**

The specified number is too large to be represented internally to the compiler, i.e. in 32 bits.

Objects that have been cast are not lvalues

An object has been cast in an lvalue context; this is illegal in ANSI C.

Only const and volatile can qualify a pointer; found ***type***

The only type qualifiers of a pointer are `const` and `volatile` but *type* was found instead.

Operand ***number*** **to** ***instruction*** **is larger than a word**

The arguments to an `__asm` load or store pseudo-instruction must fit in a machine word.

Operand ***number*** **to** ***instruction*** **is not word-sized**

The arguments to an `__asm` store pseudo-instruction must fit exactly in a machine word.

Operand to ***instruction*** **must be a constant or local variable**

An illegal operand has been given to an `__asm ldl` or `stl` instruction.

Operand to ***instruction*** **is larger than a word**

The operand to a primary instruction inside `__asm` must fit in a machine word.

Out of memory
Out of store (for error buffer)
Out of store (in cc_ alloc)

The compiler ran out of available memory.

Overlarge (single precision) floating point value found

The number is too large to represent in single word (32 bit) floating point format.

Overlarge floating point value found

The number is too large to represent in double-word (64 bit) floating point format.

Quote (*char*) inserted before newline

The specified quote character was found before a newline character. This may indicate a a spurious character or a missing closing quote.

Re-using ***struct/union*** **tag** ***identifier*** **as** ***union/struct*** **tag**

The named identifier has been used to identify two different types of object.

Size of array ***identifier*** **is unknown**

The type for objects with no linkage must be complete by the end of their definition. The identified array needs either an explicit size specified or an initializer.

Storage class ***store class*** **incompatible with** ***store class***

Two incompatible storage classes have been used in a declaration. For example, `extern static foo;` generates this error because `extern` and `static` are incompatible types.

Storage class ***store class*** **not permitted in context** ***context***

The specified storage class is not permitted in the context in which it has been used. This error would be generated, for example, if storage class `auto` were to be used at the top level.

'struct *identifier'* has no *identifier* field

The structure contains no field of that name.

'struct *identifier'* must be defined for (static) variable declaration

An undefined structure has been used in a variable declaration.

'struct *identifier'* not yet defined – cannot be selected from

A reference was made to an undefined structure.

Syntax error in command line option *string*

There is a syntax error on the command line. This is probably due to mismatched quotation marks (").

Too few operands for *instruction*

A load or store `__asm` pseudo-instruction has too few arguments.

Too few arguments to macro *identifier*(. . . on line *number*

There are too few arguments to the macro invoked on line *number.*

Too many operands for *instruction*

A load or store `__asm` pseudo-instruction has too many arguments.

Too many arguments to macro *identifier*(. . . on line *number*

There are too many arguments to the macro invoked on *number.*

Too many errors

After 100 serious errors, the compilation aborts.

Too many initialisers in {} for aggregate

An aggregate type, for example an array, has been initialized with more values than can be accommodated.

Type *type1* inconsistent with *type2*

Two incompatible type identifiers are being used in the declaration of a single object. For example, the declaration `double int x;` would generate this error.

Type disagreement for *identifier*

The specified identifier has already been assigned a different type.

Typedef name *type* used in expression context

A type definition has been used in an expression.

Type qualifier *type* not allowed to qualify *type* type

'`const`' may not be repeated in the qualifying list of a type, and similarly for '`volatile`'.

Undefined '*struct/union identifier1*' member/object: *identifier2*

The structure or union is, at present, undefined.

Uninitialised static [] arrays illegal

Static arrays of unspecified size must be initialized.

'union *identifier'* **has no** *identifier* **field**

The union contains no field of that name.

'union *identifier'* **must be defined for (static) variable declaration**

An undefined union has been used in a variable declaration.

'union *identifier'* **not yet defined – cannot be selected from**

A reference was made to an undefined union.

Unknown directive: *#identifier*

identifier is not a valid preprocessor directive. Check spelling and/or syntax.

Unknown instruction *instruction*

instruction is not a defined transputer instruction.

'void' values may not be arguments

Actual arguments in function calls cannot be of type `void`.

'while' expected after 'do' – found *string*

The `while` statement is missing from a `do . . . while` construct. *string* marks the position.

Zero width named bit-field – 1 assumed

Named bit-fields must be at least one bit wide.

2 icconf - configurer

This chapter describes the configurer tool icconf that configures code for transputer networks. It describes the command line syntax and explains how the tool generates a configuration data file from a configuration description for input to the code collector tool. The chapter ends with a list of configurer diagnostics and error messages.

2.1 Introduction

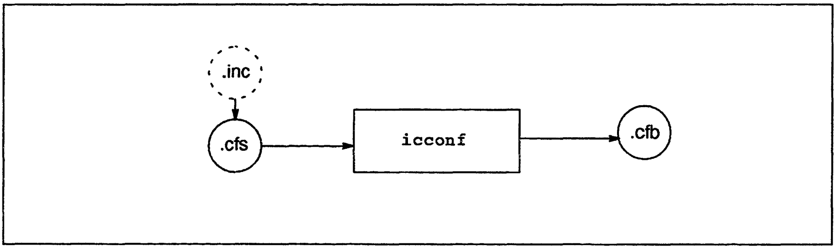
The configurer takes a *configuration description* created using the transputer configuration language and produces a *configuration binary file* which icollect uses to generate bootable code for a transputer network.

A configuration description describes how code is to be run on a network of transputers. It consists of separate definitions of the software and hardware networks, and a mapping description which defines how the software will be placed on the processor network. Using this description the configurer allocates code to particular processors and performs wide ranging consistency checks on the mapping of software to hardware.

icconf enables any topology of software network to be placed on any topology of hardware network. There are no restrictions on how many communication channels may be allocated to a single inter–processor link. Where possible channels should be left unplaced by the user, so that icconf can implement the 'best' route through the network. Processes must be allocated to specific processors and any channels going to edges must be placed on the specific edge links.

Code to be run on separate processors must be linked code. Linked units that are to be run on the same transputer must be compiled for the same or a compatible transputer type.

The operation of the configurer tool in terms of the standard toolset file extensions is illustrated below.



2.2 Configuration language implementation

The configuration language supported by `icconf` has a number of implementation characteristics of which the programmer should be aware. These are briefly listed below; details can be found in appendix F.

- Array subscript ranges are dependent on the word-length of the machine running the configurer.
- Source lines must not exceed 1024 characters. Leading and following white space is ignored.
- The number of dimensions for identifiers and array constants must not exceed 64.

2.3 Running the configurer

The configurer takes as input a configuration description file and produces a configuration data file for input to the collector tool.

To run the configurer use the following command line:

► `icconf` *filename* { *options* }

where: *filename* is the configuration description file. The filename is interpreted as given and no file extension is assumed.

options is a list of one or more options from table 2.1.

Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

Options may be supplied in an indirect argument file, prefixed by '@'. See section A.1.2 for details.

Only one filename may be given on the command line.

If no arguments are given on the command line a reduced help page is displayed giving the command syntax.

Example of use:

```
icc hello –st20
ilink –st20 hello.tco –f cstartup.lnk
icconf  hello.cfs
icollect  hello.cfb
```

```
icc –t450 hello.c
ilink  –t450 hello.tco –f cstartup.lnk
icconf  hello.cfs
icollect  hello.cfb

icc –t805 hello.c
ilink  –t805 hello.tco –f cstartup.lnk
icconf  hello.cfs
icollect  hello.cfb
```

Option	Description
C	Checks the configuration description only. No configuration data file is generated.
GA	Generates a configuration which can be debugged by the INQUEST debugger in interactive mode. See section 2.3.3. Must not be used with the NV option.
HELP	Displays a full help page which lists all the standard options.
I	Displays extra information as the tool runs.
NV	Generates a configuration without virtual routing. **Note:** any definition of `router` attributes for processor nodes will be redundant. Must not be used with the GA option. See section 2.3.4.
NWI	Disable the generation of warning messages about potential problems with the interface of processes. **Note** this option overrides the effect of the WP option when generating warnings of this type.
NWU	Disable the generation of warning messages about undefined attributes of nodes, processes and processors. **Note** that this option overrides the effect of the WP option when generating warnings of this type.
O *filename*	Specifies an output filename. If no output file is specified the configuration binary file is given the base name of the input file and the `.cfb` extension is added.
P *procname*	Specifies the name of the root processor when configuring for EPROMs. *procname* must not be an element from an array of processors.
PRE	Generates a configuration which can be profiled by the INQUEST execution profiler. See section 2.3.3.
PRU	Generates a configuration which can be profiled by the INQUEST utilization profiler. See section 2.3.3.
RA	Creates a file suitable for a boot-from-ROM application in which the user and system processes for the root processor and all other processors are loaded into RAM to execute.
RO	Creates a file suitable for a boot-from-ROM application in which the user and system processes for the root processor execute in ROM and for all other processors the user and system processes are loaded into RAM to execute.
RS *romsize*	Specifies the size of ROM on the root processor. Only valid when used with the 'RA' or 'RO' options. *romsize* is specified in decimal format and can be followed by 'K' or 'M' to indicate kilobytes or megabytes.
V	Enables the generation of configurer messages of severity *Information*.
W	Disables configurer messages of severity *Warning*.
WP	Generates additional pedantic *Warning* messages.

Table 2.1 icconf command line options

2.3.1 Default command line

Default command line parameters can be defined on the system in the ICCONFARG environment variable. Parameters must be specified using the syntax required by the configurer command line.

2.3.2 Boot from ROM options

The boot-from-ROM options 'RO' and 'RA' indicate that the program is to be collected for loading into EPROM and select the execution mode (from ROM or RAM) for the root transputer code. The 'RS' option enables the size of ROM on the root processor to be specified.

2.3.3 Support for INQUEST

Three options are available to support the use of the INQUEST debugger and profiler tools.

The GA option generates a configuration which can be debugged by the INQUEST debugger in interactive mode.

When the GA option is used, the configurer will allocate debugging kernels to all processors which have been placed with at least one process which is available for debugging. See figure 2.1.

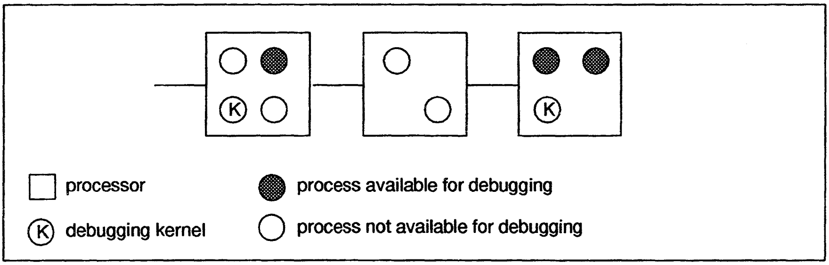

□ processor
● process available for debugging
Ⓚ debugging kernel
○ process not available for debugging

Figure 2.1 Allocation of debugging kernels using the GA option

Processes may be debugged provided the nodebug attribute is set to FALSE. When the GA option is used the default for the nodebug attribute is FALSE.

The GA option must not be used with the RO boot-from-ROM option, or with the PRE and PRU profiling options or with the NV option.

The PRE and PRU options generate a configuration which can be profiled by the INQUEST execution and utilization profilers respectively. Processes may be profiled provided the noprofile attribute is set to FALSE, which is the default. The PRE and PRU options are mutually exclusive.

2.3.4 Virtual routing processes

The configurer will automatically add virtual routing processes if they are required. If virtual routing is not required the virtual router can be disabled by using the 'NV' command line option. **Note:** the use of this option also has an affect on the value of **LoadStart**, see section 2.3.10. The NV option must not be used with the GA option.

If occam modules are being configured using icconf and the 'NV' option is to be specified then the preceding compilation and linking stages of the occam modules may be performed using the occam compiler and linker 'Y' option. This is because library i/o will not be required.

Further information about virtual routing is given in the '*User Guide*' for those toolsets which support virtual routing.

2.3.5 Mixed language programming

When the program includes a mixture or C and occam modules the configurer will perform some extra checks to ensure continuity exists. If the command line options used with icconf indicate that virtual routing is enabled then the configurer checks that library i/o is enabled in the occam modules. If it is not, a warning will be issued.

2.3.6 Configurer library files

Depending on the command line options used, the configurer reads a number of special library files which contain system processes. The library files are searched for on the directory specified by ISEARCH. This is normally the toolset libs directory, in which the files were originally installed. The library files are listed in table 2.2.

Library	Description
sysproc.lib	System startup processes for the different transputer types.
sysvlink.lib	Software through-routing processes.
sysdebug.lib	Debugging kernels to support the INQUEST debugger.
sysprof.lib	Profiling kernels to support the INQUEST profiler.

Table 2.2 Configurer library files

2.3.7 Standard include files

A number of standard include files are supplied to assist with configuration. All include files carry the .inc extension.

Defaults file setconf.inc

Configurer defaults are defined in the file setconf.inc.This file is automatically included at startup and does not need to referenced by an #include statement.

setconf.inc contains a number of boolean constants, definitions of process and processor base types, and predefined processor types. setconf.inc is supplied on the libs installation directory.

Other include files

Two other include files are provided on the libs directory. These provide definitions of processor and memory combinations for SGS-THOMSON systems products.

trams.inc	Processor type type definitions for SGS-THOMSON systems TRAnsputer Modules (TRAMs).
boards.inc	Processor type definitions for SGS-THOMSON systems transputer evaluation boards.

These two files are *not* automatically referenced by the configurer and need to be included in the normal way.

SGS-THOMSON systems products are available separately through your local distributor.

2.3.8 Configuration description examples

A series of example configuration descriptions are supplied in the icconf examples subdirectory. These include configurations for specific network topologies such as rings, grids, trees, and pipelines.

Further simple configurations are provided in the simple examples subdirectory.

2.3.9 Search paths

If a directory path is not specified the configurer uses the standard toolset search mechanism for locating input files, include files, and system library files. Briefly, the current directory is searched first, followed by the directories specified by ISEARCH (if defined on the system). For details see appendix A.

2.3.10 Default memory map

By default the configurer maps code into memory in the following order beginning at **LoadStart**: stack; code; vector space; static; heap and system data. The memory segments are contiguous. The upper limit of the memory available to the configurer is defined in the configuration description file (.cfs), by the memory attribute specified for the processor node. The default memory map is illustrated in Figure 2.2. **Note:** vector space is only required if occam modules are present in mixed language programs.

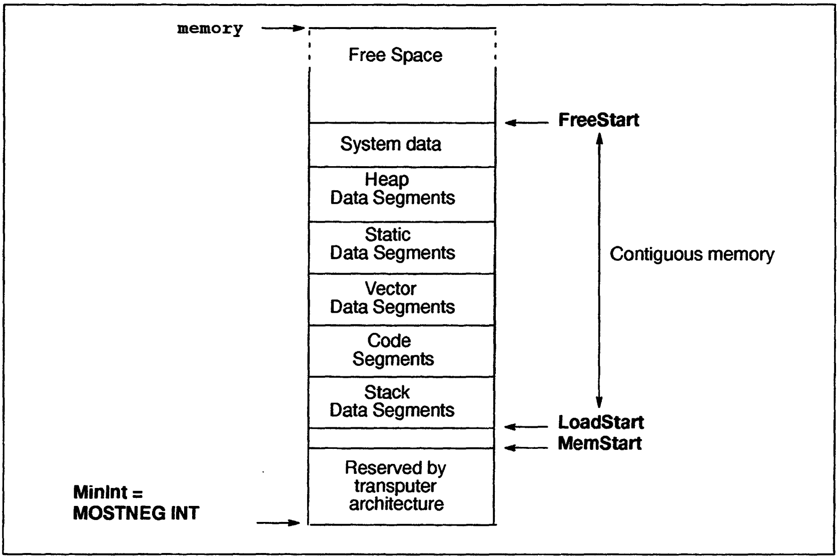


Figure 2.2 `icconf` default memory map

The first 2 or 4 Kbytes of memory (16K for the ST20450 and T450) above **MOSTNEG INT** is implemented as on–chip RAM, and includes a few words which are reserved by the transputer hardware for the implementation of links and other hardware registers. **LoadStart** is either just above or coincident with **MemStart**, see below. **Note:** that on the ST20 the start of **MemStart** is variable and is specified using the `memstart` attribute in the configuration description. **FreeStart** is the start of unused memory.

LoadStart

The position of **LoadStart** for a processor varies depending on the use of `icconf` command line options and the `reserved` processor attribute, optionally specified within a configuration description.

When the `reserved` processor attribute is specified, **LoadStart** is defined to be the memory location obtained by adding the value of `reserved` to **MOSTNEG INT**.

When the `reserved` processor attribute is *not* specified, **LoadStart** is coincident with or just above **MemStart**:

- **LoadStart** = (**MemStart** + 12 words) when the '`NV`' command line option has *not* been specified i.e. virtual through-routing support is enabled.

  Note: for most processors, **MemStart** is assumed to be greater or equal to 28 words from **MOSTNEG INT**, but if not, then **LoadStart** = **MOSTNEG INT** + 40 words. Processors which have a **MemStart** which is less than 28 words from **MOSTNEG INT** are the M212, T212, T222, T225 and T414.

- **LoadStart** = (**MemStart** + 6 words) when the 'NV' command line option is specified, disabling virtual through-routing and profiling is enabled.
- **LoadStart** = **MemStart** when the 'NV' command line option is specified but profiling is not enabled.

The value of **LoadStart** can be checked once the application has been collected, by generating and examining the collector map file, see chapter 5.

2.3.11 System processes

System processes are code and data placed by the configurer for initializing the application. By default, the system startup processes' code and data are placed into user process data areas e.g. stack, heap and so on. These system processes do not interfere with the user's data because they complete their task before the space is needed by the user's code. **Note:** these system processes do not include the virtual routing processes and profiler and debugger kernels, placed by the configurer.

2.4 Configurer messages

Errors in the configuration source produce error messages in standard toolset format. Details of the format can be found in section A.7.

Messages are generated at *Information, Warning, Error, Serious* and *Fatal* severities. Most messages are generated at *Error* severity. The configurer aborts after 400 source file errors.

In the following lists messages are grouped by severity and listed in alphabetical order.

2.4.1 Information

The following messages are generated at severity level *Information*. They are enabled by the `I` command line option.

isolated root processor *'name'*

If multiple routing sub–networks are required then the named processor has been selected as the root processor within its sub–network because it is the only processor in its sub–network. This processor will have no connection to any other processors via the basic spanning trees.

no through routing on processor *'name'*

The named processor has been prevented from being used for through-routing channels by assigning its `routecost` attribute to be `INFINITE_COST` or greater.

placed channel *'name'* **onto link** *'name'*

The named channel has been placed automatically by the configurer onto the named link.

placed edge '*name*' onto edge '*name*'

The named input (or output) edge has been placed automatically by the configurer onto the named hardware edge.

placing *string* onto processor '*name*'

The named processor has been placed with a software through-routing kernel process. *string* is the module that has been placed.

processor *'name'* I *string1* O *string2* (Routers *count1*, DeMuxes *count2*, Muxes *count3*)

The named processor requires virtual link support processes. *count1*, *count2*, and *count3* specify the number of through-routing, demultiplexing, and multiplexing modules. *string1* and *string2* describe the virtual links implemented by each input and output link of the processor.

These strings consist of pairs of characters; where the first is a digit and specifies the link number and the second, which can be `l`, `R`, or `r`, specifies if local data, local and through-routed data, or just through-routed data is carried by that link.

selected root processor *'name'*

The named processor has been selected as the root processor of the spanning tree derived to provide a basic route for all routed channels. If the application requires routing as multiple sub–networks then each sub–network will generate this message.

2.4.2 Warnings

The following messages are generated at severity level *Warning.*

attribute '*name*' definition ignored

This message can only occur in mixed language programs incorporating occam modules. The named `stacksize` or `heapsize` attribute has been assigned a value that has been ignored.

attribute '*name*' has been reassigned

Named attribute has been reassigned.

attribute '*name*' undefined

Named attribute has not been assigned a value.

Note: this message is not generated when the `NWU` option is specified and if the `NWI` option is specified then this message is not generated for the interface attributes of processes.

If the `WP` option is specified then this message is generated for all attributes that have not been assigned.

cannot through route between processors *'name1'* **and** *'name2'*

This message is output when the basic routing algorithm within the configurer cannot find a viable route between a pair of processors. This may explain subsequent errors output concerning channels running between these two processors. *name1* and *name2* identifies the processors that cannot be connected.

channel '*name'* **unconnected and unplaced**

Named channel has not been connected or placed.

Note: this message is not generated when the `NWI` option is specified

connector '*name'* **unused**

Named connector has not been used in a `connect` statement.

Note: this message is only generated when the `WP` option is specified

edge *'name'* **unconnected and unplaced**

The named input (or output) edge has not been connected or placed.

Note: this message is only generated when the `WP` option is specified

edge *'name'* **unconnected**

The named edge has not been connected.

Note: this message is only generated when the `WP` option is specified

exceeded `linkquota` on processor *'name'*; *count1* **inputs** *count2* **outputs**

This warning is output if the requested `linkquota` on any processor has been exceeded by the configurer. *name* identifies the processor concerned while *count1* and *count2* indicates the number of input and output links that are required.

illegal definition for attribute *'name'* **when profiling**

The named attribute has been assigned an illegal value. This is generated if a process has been specified to start in high priority and the `PRE` or `PRU` options have also been specified.

insufficient memory size for attribute *'name'*, *value*

The memory size specified by the named attribute is insufficient for the type of processor. This will be generated if the `memory` or `reserved` processor attributes have been assigned a memory size that is less than **LoadStart**. *value* is the memory size.

link '*name*' unconnected

Named link has not been connected.

Note: this message is only generated when the `WP` option is specified

nested comment statements, *value*

One or more nested comments have been found by the configurer. *value* is the number of nested comments found.

no Inquest debugger kernels to place

All processes have had their `nodebug` attribute set to `TRUE` (and therefore no processors are available for debugging) .

overflow in hexadecimal escape character

A numerical overflow has occurred during the evaluation of a hexadecimal escape character whose range is from 0 to 255.

overflow in octal escape character

A numerical overflow has occurred during the evaluation of an octal escape character whose range is from 0 to 255.

processor '*name*' unconnected

Named processor has not been connected to the network.

processor '*name*' unused

Named process has been connected to the network but has had no user processes placed onto it.

process '*name*' using direct channel input/output

The named process has been compiled to use direct instructions instead of indirect functions for channel i/o and because the process may also execute with the virtual link support processes, an incompatibility may arise.

process '*name*' using indirect channel input/output

The named process has been compiled to use indirect functions instead of direct instructions for channel i/o and because the process will not be executed with the virtual link support processes, the use of functions for channel i/o is unnecessary. **Note:** this message is only generated when the `WP` option is specified.

unable to debug on processor '*name*' as not accessable from root processor

The named processor cannot be debugged by the INQUEST debugger because there exists no route, either physical or virtual, between the named processor and the root processor (which is connected to the host).

using single hop software virtual links

This is output if the configurer adds any run-time multiplexing software to the user's program to support virtual channels.

Note: this message is only generated when the `WP` option is specified

using through routed software virtual links

This is output if the configurer adds any run-time multiplexing and through-routing software to the users program to support virtual channels.

Note: this message is only generated when the `WP` option is specified

string **for process** '*name*' **overlaps memory registers**

A memory segment of the named process overlaps the hardware registers, which are the memory locations below **MemStart**, of the processor that the process has been placed on. *string* is the memory segment name which can be `code`, `heap`, `stack`, `static` or `vector`.

string **of process** '*name*' **overlaps** *string* **of process** '*name*'

A memory segment of the first named process overlaps a memory segment of the second named process. *string* is the memory segment name which can be `code`, `heap`, `stack`, `static` or `vector`.

2.4.3 Errors

The following messages are generated at severity level *Error*. Most configurer diagnostics are generated at this level.

attribute '*name*' cannot be reassigned

The named attribute cannot be reassigned. This is only generated for the `element` attribute for nodes, the `type` attribute for processors and the `numlinks` attribute for processors with `type` set to "`ST20`".

attribute '*name*' multiply defined in '*name*'

Named attribute has been declared within the `interface` attribute and its name clashes with a previously declared attribute or its name clashes with the name of a predefined attribute for the named process.

attribute ‘*name*′ undefined in ‘*name*′

Named attribute is an undefined attribute of the named symbol.

attribute ‘*name*′ undefined

Named attribute has not been assigned a value which is required.

automatic Tree Router provoked Deadlock

A potential communications deadlock has been detected in the routing network constructed by the virtual through-routing algorithm where a cycle of links that may through-route to each other has been created. This is an internal consistency error and should never be generated.

bad channel placement between processors ’*name*’ and ’*name*’

A channel placement between the named processors specifies link numbers which do not correspond to each end of the same link connection. This is an internal consistency error and should never be generated.

cannot route channel between processors ’*name1*’ and ’*name2*’

This message indicates a failure to complete auto–routing. The message is output each time a channel cannot be implemented either because it is too long (> 24 hops) or no viable route exists to support it. *name1* and *name2* are source and destination processors between which the data or acknowledge path of a channel cannot be routed.

channel ‘*name*′ connected and unplaced

The named channel has been left unplaced because the configurer was unable to automatically place the channel.

This is generated if either the other end of the connection is an input (or output) software edge which has not been placed or if the other end of the connection has been placed onto an unconnected link (or hardware edge).

channel *‘name1’* connected and unplaced, invalid placement of ‘*name2*′

The channel ‘*name1* has been left unplaced because the configurer was unable to automatically place the channel ‘*name1*’.

This is generated if the other end of the connection ‘*name2*′ has been placed onto a link (or hardware edge) which is connected to a processor different to that on which the channel *name1* is to reside.

channel ‘*name*′ multiply connected

Named channel has been used more than once in a `connect` statement.

channel '*name*' multiply placed

Named channel has been used more than once in a `place` statement.

channel '*name*' placed onto link *'name1'*, expecting connection to *'name2'*

The named channel has been placed onto the link *name1* where the link *name1* has not been connected appropriately. It is expected that the link *name1* is connected to the link (or hardware edge) *name2*.

This is generated by both ends of a software connection being placed onto links or hardware edges (as appropriate) which are not connected.

channel '*name*' placed onto unconnected link *'name'*

The named channel has been placed onto the named link which has not been connected

channel '*name*' unconnected and placed

Named channel has been placed and has not been connected.

connect '*name*' to '*name*' illegal, both channels

An illegal `connect` statement has been specified where both named elements are channels and they communicate in the same direction.

connect '*name*' to '*name*' illegal, both edges

Connect statement is illegal because the named elements are both edges.

connect '*name*' to '*name*' illegal, channel/edge

An illegal `connect` statement has been specified where the first named element is a channel and the second named element is an input (or output) edge and they communicate in different directions.

connect '*name*' to '*name*' illegal, edge/channel

An illegal `connect` statement has been specified where the first named element is an input (or output) edge and the second named element is a channel and they communicate in different directions.

connector '*name*' multiply placed

Named connector has been used more than once in a `place` statement.

connector '*name*' multiply used

Named connector has been used more than once in a `connect` statement.

constant dimension sizes inconsistent, *value*

A constant array has been defined which has inconsistent dimension sizes for some of its elements. *value* is the number of the incorrect dimension, counting from zero.

constant dimensions incompatible with '*name***'**

Named symbol has been assigned a constant value whose dimensions are incompatible with those of the symbol.

constant element types not equal, *type*

A constant array has been defined where some or all of its elements have non-equal types. *type* is the expected type for each element in the array.

constant type incompatible with '*name***',** *type*

Named symbol has been assigned a constant value whose type is incompatible with that of the symbol. *type* is the expected type for the constant.

edge '*name***' connected and unplaced**

The named input (or output) software edge has been left unplaced since the configurer was unable to automatically place the input (or output) software edge.

This is generated if either the other end of the connection has not been placed or if the other end of the connection has been placed onto an unconnected link.

edge '*name1***' connected and unplaced, invalid placement of '***name2***'**

The input (or output) software edge '*name1*' has been left unplaced because the configurer was unable to automatically place the input (or output) software edge '*name1*'.

This is generated if the other end of the connection '*name2*' has been placed onto a link which is connected to a hardware edge different to that on which the input (or output) software edge '*name1*' is to reside.

edge '*name***' multiply connected**

The named edge has been specified more than once in a `connect` statement.

edge '*name***' multiply placed**

The named edge has been specified more than once in a `place` statement.

edge '*name***' placed onto edge '***name1***', expecting connection to '***name2***'**

The named input (or output) software edge has been placed onto the hardware edge *name1* where the hardware edge *name1* has not been connected appropriately. It is expected that the hardware edge *name1* is connected to the link *name2*.

This is generated by both ends of a software connection being placed onto links or hardware edges (as appropriate) which are not connected.

edge '*name*' placed onto unconnected edge '*name*'

The named input (or output) software edge has been placed onto the named hardware edge which has not been connected

edge '*name*' unconnected and placed

The named edge has been placed and has not been connected.

element '*name*' in connection undefined

The named element has been used in a `connect` statement and is undefined. This is only generated from process and processor types.

element '*name*' in placement undefined

The named element has been used in a `place` statement and is undefined. This is only generated from process and processor types.

element '*name*' not completely subscripted

Named symbol has been defined as an array and has not been completely subscripted.

host edge '*name*' unconnected

The named host edge has not been connected to anything when booting from link.

host edge '*name*' undefined

When configuring to boot from link, the named host edge has not been declared in the configuration source. This error can only be caused if the standard include file `setconf.inc` has been altered.

illegal # directive type, found '*string*'

An illegal identifier for a #directive has been specified. *string* is the illegal directive identifier.

illegal 16 bit RAM + ROM memory size for processor '*name*', *value*

The named processor is a 16 bit processor which has been specified RAM and ROM memory sizes whose total is illegal for 16 bit processors. *value* is the illegal memory size.

illegal 16 bit RAM memory size for attribute '*name*', *value*

The named processor is a 16 bit processor which has been specified a RAM memory size which is illegal for 16 bit processors. *value* is the illegal memory size.

illegal 16 bit ROM memory size for processor '*name*', *value*

The named processor is a 16 bit processor which has been specified a ROM memory size which is illegal for 16 bit processors. *value* is the illegal memory size.

illegal 16 bit address for attribute '*name*'

The named attribute, which is a sub-attribute of the `location` attribute for a process, has been assigned an address which is illegal for 16 bit processors. *value* is the illegal address.

illegal assignment for attribute '*name*'

The named attribute has been specified in an attribute modification statement and is not of arithmetic type.

illegal definition of attribute '*name*' for PRI PAR process

The named attribute, which is always the `priority` attribute of a process, has been assigned to `HIGH` when the code for the process has been specified to also execute at high priority.

illegal definition of attribute '*name*' when executing from ROM

The named attribute, which is always the `code` attribute of the `location` attribute for a process, has been assigned an address when the code for the process is to execute from ROM.

illegal dimension size, *value*

A dimension size not greater than zero has been specified. *value* is the dimension number with the illegal dimension size.

illegal escape character sequence, *char*

An illegal escape character sequence has been specified. *char* is the illegal escape character.

illegal format character constant, *char*

An illegal format character constant has been specified. *char* is the unexpected character found in the character constant.

illegal format hexadecimal constant, *char*

An illegal format hexadecimal constant has been specified. *char* is the unexpected character found in the hexadecimal constant.

illegal memory size for attribute '*name*', *value*

The named attribute, which is always the `reserved` attribute of a processor, has been assigned a memory size which is greater than the size assigned to the `memory` attribute of the processor. *value* is the illegal memory size.

illegal number of dimensions for edge '*name*', *value*
illegal number of dimensions for processor '*name*', *value*

The named identifier has been declared as an array when it should have been declared as a scalar. This is either generated for the host edge (when booting from link) or for the root processor (when booting from ROM). *value* is the number of declared dimensions.

illegal number of dimensions, *value*

Number of dimensions for a symbol or constant exceeds the maximum number of dimensions allowed by the configurer. *value* is the maximum number of dimensions allowed.

illegal number of subscripts for '*name*', *value*

Number of subscripts specified for the named symbol exceeds the number the symbol requires. *value* is the maximum number of subscripts allowed.

illegal number of subscripts for constant, *value*

Number of subscripts specified for a constant exceeds the number the constant requires. *value* is the maximum number of subscripts allowed.

illegal operation for attribute '*name*'

The named attribute has been used inappropriately in an attribute modification statement.

illegal source file character, *value*

An unexpected character has been found in the source file. *value* is the ASCII value for the illegal character.

illegal subscript value, *value*

A subscript value of less than zero or greater than the dimension size has been specified. *value* is the number of the dimension with the illegal subscript value.

illegal token for expression, found *token*

An unexpected token has been found at the start of an expression. *token* is the unexpected token.

illegal token for statement, found *token*

An unexpected token has been found at the start of a statement. *token* is the unexpected token.

illegal type for '*name*' in USE statement, *type*

Named symbol has been specified in a `use` statement and is not a process or a process type. *type* is the type of the symbol.

illegal type for '*name*' in connection, *type*

Named symbol has been specified in a `connect` statement and is not a channel, edge, link or connector. *type* is the type of the symbol.

illegal type for '*name*' in definition, *type*

Named symbol has been specified in a node definition statement and is not a node type. *type* is the type of the symbol.

illegal type for '*name*' in expression, *type*

Named symbol has been specified in an expression and is not a constant value. *type* is the type of the symbol.

illegal type for '*name*' in modification, *type*

Named symbol has been specified in an attribute modification statement and is not a node. *type* is the type of the symbol.

illegal type for '*name*' in placement, *type*

Named symbol has been specified in a `place` statement and is not a process, processor, edge, channel, link or connector. *type* is the type of the symbol.

illegal type for '*name*', expecting *type1* **found** *type2*

The named identifier has been declared with an unexpected type. The named identifier will either be the string specified by the `P` option and has not been declared as a processor (when booting-from-ROM) or is the host edge and has not been declared as an edge (when booting-from-link). *type1* is the expected type for the identifier and *type2* is the actual type of the identifier.

illegal type for IF statement condition, *type*

The condition value for an `if` statement is not of integral type. *type* is the type of the condition value.

illegal type for arithmetic operator *operator, type*

The operand of an arithmetic unary operator is not of arithmetic type. *type* is the type of the operand and *operator* is the arithmetic operator.

illegal type for boolean operator *operator, type*

The operand of a boolean binary operator is not of integral type. *type* is the type of the operand and *operator* is the boolean operator.

illegal type for condition operator *operator, type*

The condition value for a conditional ternary operator is not of integral type. *type* is the type of the condition value and *operator* is the conditional operator.

illegal type for connector '*name***' in placement,** *type*

Named symbol is a connector defining a connection and has been used in the incorrect position in a `place` statement. *type* is the type of connection defined by the symbol.

illegal type for dimension size, *type*

The type of a dimension size value is not of integral type. *type* is the type of the dimension size value.

illegal type for integral operator *operator, type*

The operand of an integral unary operator is not of integral type. *type* is the type of the operand and *operator* is the integral operator.

illegal type for module in USE statement, *type*

The module specified in a `USE` statement is not of string type. *type* is the type of the module.

illegal type for subscript value, *type*

The type of a subscript value is not of integral type. *type* is the type of the subscript value.

illegal type for value in REP statement, *type*

The base or limit value for a replicator statement is not of integral type. *type* is the type of the base or limit value.

illegal types for arithmetic operator *operator, type1* **and** *type2*

The operands of an arithmetic binary operator are not both of arithmetic type. *type1* and *type2* are the types of the operands and *operator* is the arithmetic operator.

illegal types for equality operator *operator, type1* **and** *type2*

The operands of an equality binary operator are not both of arithmetic type. *type1* and *type2* are the types of the operands and *operator* is the equality operator.

illegal types for integral operator *operator, type1* **and** *type2*

The operands of an integral binary operator are not both of integral type. *type1* and *type2* are the types of the operands and *operator* is the integral operator.

illegal use of constant for element

A constant value has been used as an element.

illegal use of subfield operator for '*name*'

The named symbol, which has no accessible attributes, has been used with the subfield operator.

illegal use of subfield operator for constant

A constant value has been accessed using the subfield operator.

illegal value for attribute '*name*'

Named attribute has been given a value which is inconsistent with the type of the attribute and its semantic meaning.

incompatible interface, attribute '*name*' has different type, *type*
incompatible interface, attribute '*name*' has referenced type, *type*
incompatible interface, attribute '*name*' has unequal dimensions
incompatible interface, process '*name*' has too few parameters
incompatible interface, process '*name*' has too many parameters

These messages can only be generated in mixed language programs incorporating occam modules. The named symbol is an occam process and the `interface` defined for the process mismatches the formal parameter list defined in the object file associated with the process in a `use` statement.

insufficient RAM memory for processor '*name*', *value* bytes amiss

Named processor's total RAM memory size is insufficient for the number of processes placed on the processor (which includes their data requirements). *value* is the number of extra bytes needed to accommodate all the processes on the processor.

link '*name*' multiply connected

Named link has been used more than once in a `connect` statement.

link '*name*' multiply placed

Named link has been used more than once in a `place` statement.

missing (for SIZE operator, found *token*

The `size` operator has been found and an opening parenthesis was expected to be found after the keyword `size`, instead of which the token *token* was found.

missing) for SIZE operator, found *token*

The `size` operator has been found and a closing parenthesis was expected to be found after the operand to the operator, instead of which the token *token* was found.

missing) for attribute list, found *token*

An attribute list has been found and a closing parenthesis was expected to terminate the list, instead of which the token *token* was found.

missing) for cast operator, found *token*

A cast operator has been found and a closing parenthesis was expected to be found after the type identifier, instead of which the token *token* was found.

missing) for expression, found *token*

A parenthesized expression has been found and a closing parenthesis was expected to be found after the sub-expression, instead of which the token *token* was found.

missing , or TO for CONNECT statement, found *token*

A `connect` statement has been found and a comma or the keyword `to` were expected to be found, instead of which the token *token* was found.

missing : for conditional operator, found *token*

A conditional operator has been found and a colon was expected to be found after the first sub-expression, instead of which the token *token* was found.

missing ; for statement, found *token*

A statement has been found which expects a semicolon to terminate it, instead of which the token *token* was found.

missing = for REP statement, found *token*

A replicator statement has been found and an equals was expected to be found after the replicator identifier, instead of which the token *token* was found.

missing = or (for attribute, found *token*

An attribute definition has been found and an equals or opening parenthesis were expected to be found after the attribute identifier, instead of which the token *token* was found.

missing FOR for USE statement, found *token*

A `use` statement has been found and the keyword `for` was expected to be found, instead of which the token *token* was found.

missing ON for PLACE statement, found *token*

A `place` statement has been found and the keyword `on` was expected to be found, instead of which the token *token* was found.

missing TO or FOR for REP statement, found *token*

A replicator statement has been found and the keywords `to` or `for` were expected to be found, instead of which the token *token* was found.

missing] for subscript, found *token*

A subscript operator has been found and a closing square bracket was expected to be found after the subscript value, instead of which the token *token* was found.

missing attributes for attribute list

An attribute list has been found which is empty.

missing constants for constant list

A constant list has been found which is empty.

missing identifier for # directive, found *token*

A # directive has been specified and an identifier was expected to be found after the `#`, instead of which the token *token* was found.

missing identifier for REP statement, found *token*

A replicator statement has been found and an identifier was expected to be found after the keyword `rep`, instead of which the token *token* was found.

missing identifier for VAL statement, found *token*

A value statement has been found and an identifier was expected to be found after the keyword `val`, instead of which the token *token* was found.

missing identifier for attribute list, found *token*

An attribute list has been found and an identifier was expected to be found after the opening parenthesis starting the list, instead of which the token *token* was found.

missing identifier for attribute, found *token*

An attribute list has been found and an identifier was expected to be found in the attribute list, instead of which the token *token* was found.

missing identifier for name, found *token*

A name expression has been found and an identifier was expected to be found at the start of the expression, instead of which the token *token* was found.

missing identifier for subfield, found *token*

A subfield expression has been found and an identifier was expected to be found after the subfield operator, instead of which the token *token* was found.

missing statements for statement list

A statement list has been found which is empty.

missing string for #INCLUDE statement, found *token*

An `#include` statement has been found and a string was expected to be found after `#include`, instead of which the token *token* was found.

missing type for DEFINE statement, found *token*

A define statement has been found and a type identifier was expected to be found after the keyword `define`, instead of which the token *token* was found.

missing type for attribute, found *token*

A parameter list declaration has been found and a parameter type was expected to be found in the list, instead of which the token *token* was found.

missing } for constant list, found *token*

A constant list has been found and a closing brace was expected to terminate the list, instead of which the token *token* was found.

missing } for statement list, found *token*

A statement list has been found and a closing brace was expected to terminate the list, instead of which the token *token* was found.

modification of '*name*' illegal, already used

The named symbol is a node type that has been used to derive other symbols and an attempt has been made to modify one of its attributes.

object file for process '*name*' undefined

Named process has not been associated with an object file.

overflow in REP statement expression

A numerical overflow has occurred during the evaluation of a replicator statement, that is, the replicator identifier has overflowed.

overflow in arithmetic expression

A numerical overflow has occurred during the evaluation of an arithmetic expression.

overflow in decimal integer constant

A numerical overflow has occurred during the conversion of a string representing a 32 bit decimal integer constant.

overflow in dimension size expression

A numerical overflow has occurred during the evaluation of a dimension size expression (which is done to the precision of the hosts integer word length).

overflow in dimension sizes for '*name*'

A numerical overflow has occurred during the evaluation of the array size for the named symbol (performed to the precision of the integer word length of the host).

overflow in dimension sizes for constant

A numerical overflow has occurred during the evaluation of the array size for a constant array (performed to the precision of the integer word length of the host).

overflow in hexadecimal integer constant

A numerical overflow has occurred during the conversion of a string of digits representing a 32 bit signed hexadecimal integer constant.

overflow in octal integer constant

A numerical overflow has occurred during the conversion of a string of digits representing a 32 signed bit octal integer constant.

overflow in real double constant

A numerical overflow has occurred during the conversion of a string of digits representing a 64 bit real constant.

overflow in real float constant

A numerical overflow has occurred during the conversion of a string of digits representing a 32 bit real constant.

overflow in subscript value expression

A numerical overflow has occurred during the evaluation of a subscript value expression (which is done to the precision of the hosts integer word length).

place '*name*' on '*name*' illegal, channel/edge

An illegal `place` statement has been specified where the first named element is a channel and the second named element is an input (or output) edge.

place '*name*' on '*name*' illegal, edge/link

An illegal `place` statement has been specified where the first named element is an input (or output) edge and the second named element is a link.

process '*name*' and channel '*name*' placed on different processors

The named channel, which is a channel of the named process, has been placed on the link of a processor which is different to the processor placed with the process.

process '*name*' and processor '*name*' execution types mismatch

The named process has an execution type (specified in the object file associated with the process) which is incompatible with the execution types of other processes executing on the named processor.

process '*name*' and processor '*name*' processor types mismatch

The named process has a processor type (specified in the object file associated with the process) which is incompatible with the processor type of the named processor.

process '*name*' multiply USEd

Named process has been used more than once in a `use` statement.

process '*name*' multiply placed

Named process has been used more than once in a `place` statement.

process '*name*' unplaced

Named process has not been placed.

process type '*name*' multiply USEd

Named process type has been used more than once in a `use` statement.

processor '*name*' unconnected and placed

The named processor has not been connected to the hardware network and has been placed with one or more processes.

reference to undefined symbol '*name*'

Named symbol has been referenced but had not been defined at the point of reference.

root processor '*name*' undefined

When configuring to boot from ROM, the named processor (specified using the `P` option) has not been defined in the configuration source.

subscript out of range for '*name*', *value*

Named symbol has been accessed with the subscript operator and the subscript value used is outside the valid range of the dimension being subscripted. *value* is the dimension number that was subscripted.

subscript out of range for constant, *value*

A constant value has been accessed with the subscript operator and the subscript value used is outside the valid range of the dimension being subscripted. *value* is the dimension number that was subscripted.

symbol '*name*' multiply defined in symbol table

Named symbol has been multiply defined in the configuration source.

unable to debug on processor *'name'* as breakpoints not supported, *target*

The named processor (which has been placed with debuggable processes) cannot be debugged by the INQUEST debugger because there does not exist instruction level support on the processor to allow breakpoints to be set. *target* is the processor type.

unable to place channel *'name'* onto processor *'name'*, expecting connection to processor *'name2'*

The named channel cannot be automatically placed onto the processor *name1* by the configurer as no connection exists between itself and the processor *name2*.

This is generated if one end of a channel requires placing onto the link of a processor and there is no connection between that processor and the processor on which the other end of the channel is to reside.

unable to place channel *'name'* onto processor *'name'*, insufficient connections to processor *'name2'*

The named channel cannot be automatically placed onto the processor *name1* by the configurer because insufficient connections exists between itself and the processor *name2*.

This is generated if one end of a channel requires placing onto the link of a processor and there are not enough connections between that processor and the processor on which the other end of the channel is to reside.

unable to support Inquest debugger, host edge '*name*' not connected to root processor '*name*'

The named host edge has not been connected to the named root processor when the GA option has been specified. The named host edge must be connected to the named root processor in order for the INQUEST debugger to operate.

unable to support Inquest debugger, host edge '*name*' unconnected

The named host edge has not been connected to anything when the GA option has been specified. The named host edge must be declared and connected to the root processor in order for the INQUEST debugger to operate.

unable to support Inquest debugger, host edge '*name*' undefined

The named host edge has not been declared when the GA option has been specified. The named host edge must be declared and connected to the root processor in order for the INQUEST debugger to operate.

unalligned address for attribute *'name'*

The named attribute, which is a sub-attribute of the `location` attribute for a process, has been assigned an address which is not word aligned. *value* is the unaligned address.

uninitialised symbol '*name*' in expression

Named symbol, which is of arithmetic type, has been used in an expression and has not been assigned any value.

unterminated character constant

A character constant has been specified where a closing quote has not been found before the end of the line.

unterminated comment statement

A comment has been started and has not been terminated before the end of the file.

unterminated string constant

A string constant has been specified where a closing double quote has not been found before the end of the line.

unused connector '*name*' in placement

Named connector has not been used in a `connect` statement and has been used in a `place` statement.

value for attribute '*name*' out of range

Named attribute has been assigned a value that is not in the valid range for the attribute.

zero length character constant

A zero length character constant has been specified.

string **for process** *'name'* **exceeds maximum memory address**

The named segment of the named process has been specified an address which results in the segment exceeding the maximum memory address of the processor that the process has been placed on. *string* is the memory segment name which can be `code`, `heap`, `stack`, `static` or `vector`.

string **for process** *'name'* **overlaps ROM memory**

The named segment of the named process has been specified an address which results in the segment overlapping the ROM memory region of the processor that the process has been placed on. *string* is the memory segment name which can be `code`, `heap`, `stack`, `static` or `vector`.

string **for process '***name***' overlaps unusable memory**

The named segment of the named process has been specified an address which results in the segment overlapping the unusable memory region of the processor that the process has been placed on. *string* is the memory segment name which can be `code`, `heap`, `stack`, `static` or `vector`.

2.4.4 Serious messages

The following diagnostic messages are generated at severity level *Serious*.

ROM memory size required when booting from ROM

The `RA` or `RO` command line options have been specified and the `RS` option has been omitted.

TCOFF descriptor, illegal dimension size, *value*
TCOFF descriptor, illegal type for *name*, *type*
TCOFF descriptor, missing (, found *char*
TCOFF descriptor, missing), found *char*
TCOFF descriptor, missing :, found *char*
TCOFF descriptor, missing ? or !, found *char*
TCOFF descriptor, missing], found *char*
TCOFF descriptor, missing OCCAM PROC keyword
TCOFF descriptor, missing OCCAM identifier
TCOFF descriptor, missing OF for CHAN or PORT parameter
TCOFF descriptor, overflow in dimension size
TCOFF descriptor, undefined channel parameter
TCOFF descriptor, unknown OCCAM parameter type
TCOFF descriptor, unknown OCCAM process type
TCOFF format, expected INDEX-ENTRY command (*value*)
TCOFF format, expected LIB-INDEX-START command (*value*)
TCOFF format, expected LINKED-UNIT command (*value*)
TCOFF format, expected START-MODULE command (*value*)
TCOFF format, invalid ADJUST-POINT adjust size (*value*)
TCOFF format, invalid ADJUST-POINT value type (*value*)
TCOFF format, invalid DEFINE-MAIN symbol reference (*value*)
TCOFF format, invalid DEFINE-MAIN/DESCRIPTOR definitions
TCOFF format, invalid DEFINE-SYMBOL symbol reference (*value*)
TCOFF format, invalid DEFINE-SYMBOL value type (*value*)
TCOFF format, invalid DESCRIPTOR language type (*value*)
TCOFF format, invalid DESCRIPTOR scalar size (*value*)
TCOFF format, invalid DESCRIPTOR string size (*value*)
TCOFF format, invalid DESCRIPTOR symbol reference (*value*)
TCOFF format, invalid DESCRIPTOR vector size (*value*)

TCOFF format, invalid INDEX-ENTRY attributes (*value, value*)
TCOFF format, invalid INDEX-ENTRY language type (*value*)
TCOFF format, invalid INDEX-ENTRY string size (*value*)
TCOFF format, invalid LOAD-TEXT text size (*value*)
TCOFF format, invalid ORIGIN SYMBOL format ('*string*')
TCOFF format, invalid SET-LOAD-POINT symbol reference (*value*)
TCOFF format, invalid START-MODULE attributes (*value, value*)
TCOFF format, invalid START-MODULE language type (*value*)
TCOFF format, invalid SYMBOL string size (*value*)
TCOFF format, invalid code entry offset (*value*)
TCOFF format, multiple DEFINE-MAIN commands
TCOFF format, multiple DESCRIPTOR commands
TCOFF format, multiple LOAD-TEXT commands
TCOFF format, multiple ORIGIN SYMBOL commands
TCOFF format, multiple VIRTUAL SECTION commands
TCOFF format, undefined DEFINE-MAIN definition
TCOFF format, unexpected ADJUST-POINT command
TCOFF format, unexpected command (*value*)

An error has been detected in an object file specified by a `use` statement or a library file containing the system processes.

execution and utilisation profiling are incompatible

The PRE option and the PRU option have been specified together.

illegal ROM memory size, *value*

Value specified for the RS option is not greater than zero. *value* is the illegal memory size.

illegal command line option, *string*

An illegal option has been specified on the command line. *string* is the illegal option.

illegal format ROM memory size, *string*

An illegal format memory size value has been specified for the RS option. *string* is the illegal format memory size.

illegal record length (*value***)**

A record length has been input from a file which exceeds the maximum string length for a file. *value* is the illegal record length found.

illegal string length (*value***)**

A string length has been input from a file which exceeds the maximum record length for a file. *value* is the illegal string length found.

illegal syntax in indirect file, *string*

An argument with illegal syntax has been specified in an indirect file for the command line. *sting* is the argument with illegal syntax.

Inquest debugging and profiling are incompatible

The GA option and the PRE or PRU options have been specified together.

Inquest debugging requires software through routing

The GA option and the NV option have been specified together.

interactive and postmortem debugging are incompatible

The G option and the GP option have been specified together.

internal token buffer overflow, *value*

An internal buffer used for storing a source line has overflowed. *value* is the size of the internal buffer in bytes.

module entry *'string'* **not found in library (***type, mode***)**

The named module entry (of the specified processor type and execution mode) has been requested from a library file of system processes and has not been found. *target* is the processor type and *mode* is the execution mode.

multiple ROM memory sizes, *string*

The RS option has been specified more than once. *string* is the latest value for the RS option.

multiple input file names, *string*

The input file name has been specified more than once. *string* is the latest input file name.

multiple output file names, *string*

The O option has been specified more than once. *string* is the latest value for the O option.

multiple processor names, *string*

The P option has been specified more than once. *string* is the latest value for the P option.

processor name required when booting from ROM

The RA or RO options have been used and no P option has been specified.

running from ROM and Inquest/post mortem debugging are incompatible

The RO option and the GA option have been specified together.

too many errors occurring, *value*

Number of errors exceeds maximum number allowed. *value* is the maximum number of errors allowed.

unable to allocate memory
unable to reallocate memory

Amount of memory available to the configurer is insufficient for configuring the configuration source.

unable to close *'string'* **(***value***)**
unable to close (*value***)**
unable to open *'string'* **(***value***)**
unable to open (*value***)**
unable to read (*value***)**
unable to seek (*value***)**
unable to tell (*value***)**
unable to write (*value***)**

These messages are generated as a result of an error occurring in the host file system. *value* is the error failure code.

unable to read environment variable, *string*

An attempt has been made to read an environment variable which does not exist. *string* is the environment variable name.

unable to read indirect file, *'string'*

An error has been detected in the host filing system while reading an indirect file for the command line. *string* is the indirect file name.

unexpected end of input

The end of the file has been found unexpectedly in an object file.

2.4.5 Fatal errors

Any fatal errors which occur should be reported to your local SGS-THOMSON distributor or field applications engineer.

The following errors are generated at severity *Fatal*:

did not find all processors in routine BuildDataStructs
did not find all processors in routine FillInKernelTable
did not find all processors in routine PlaceDebugKernels

An internal error has occurred in the configurer. The configurer has found an internal inconsistency while virtual routing.

problem in allocation routines

An internal error has occurred in the configurer. The configurer has incorrectly attempted to allocate memory from the heap.

problem in deallocation routines

An internal error has occurred in the configurer. The configurer has incorrectly attempted to return memory to the heap.

3 oc - occam 2 compiler

This chapter describes the syntax and command line options of the occam 2 compiler oc. The chapter ends with a list of error messages.

The '*occam 2 Toolset Language and Libraries Reference Manual*' describes some technical aspects of occam's implementation on the transputer, including the allocation of memory, the machine representation of occam types, and other hardware dependencies.

3.1 Introduction

The toolset compiler oc implements the occam 2 language generating code for a particular transputer, transputer type or class. A target should be specified for all compilations. Transputer targets are discussed in detail in Appendix B.

oc supports some extensions to the occam 2 language, including: compiler directives, extended channel handling, extended syntax, and low level programming support. These are compiler-dependent and do not extend the definition of the language. Extensions supported by oc are listed in appendices of the '*occam 2 Toolset Language and Libraries Reference Manual*'.

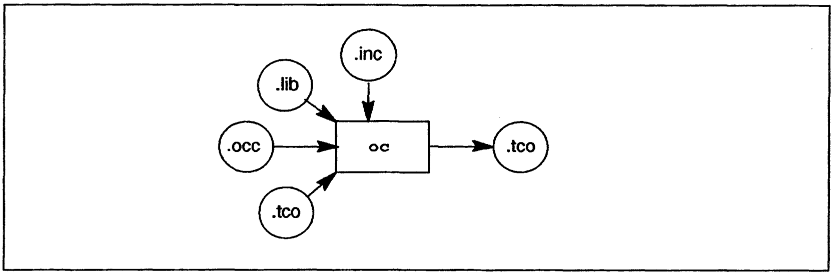
Each compilation of a program must be targetted at a specific transputer type or class and in one of three execution error modes. In addition the selection or not of the compiler's 'Y' option determines the method of channel input/output used by the compiler.

All components of a program to be run on the same transputer must be compiled for compatible target processors, error modes, and method of channel i/o.

Libraries and separately compiled units must be compiled before any file which references them can itself be compiled. It is the programmer's responsibility to ensure all components of a program are compiled in the correct order and that object code is kept up to date with changes in the source; the linker will object if this is not done. This may be assisted by using a MAKE program in conjunction with the imakef tool. The imakef tool depends on a particular system of file extensions being used. For details of version control using MAKE programs and the imakef tool see Chapter 12. The operation of the compiler in terms of standard file extensions is shown below.

occam source files can contain references to object code libraries, occam source to be included in the compilation, separately compiled occam code, and code produced by compatible compilers for other languages.

For a full description and formal definition of the occam 2 language see the '*occam 2 Reference Manual*'.



The object file is generated by the compiler in *T*ransputer *C*ommon *O*bject *F*ile *F*ormat (TCOFF). Object files are required to be in this format to be compatible with other tools in the toolset such as the librarian and linker tools.

3.2 Running the compiler

The occam 2 compiler takes as input an occam source file and compiles it into a binary object file. Command line options determine the target transputer for the compilation, the compilation error mode, and other compiler facilities such as alias and usage checking. A target processor and compilation error mode should be specified for each compilation. By default the compiler produces code in HALT mode.

To invoke the compiler use the following command line:

```
▶    oc  filename  {options}
```

where: *filename* is the name of the file containing the source code. If you do not specify a file extension, the extension .occ is assumed.

options is a list, in any order, of one or more of the options given in Table 3.1.

> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

If the compilation is unsuccessful, error messages are displayed giving the name of the file and the number of the line where the error occurred. Compiler error messages are listed in section 3.14.

Examples:

```
oc —t450 simple.occ
ilink -t450 simple.tco hostio.lib -f occam450.lnk
occonf simple.pgm
icollect simple.cfb

oc —t805 simple.occ
ilink -t805 simple.tco hostio.lib -f occama.lnk
occonf simple.pgm
icollect simple.cfb
```

In each of the above examples the source code is compiled for a single processor. The examples also shows the commands for linking, configuring, collecting, and loading the program.

Option	Description
Transputer type	See Appendix B for transputer type options.
A	Prevents the compiler from performing alias checking.This option also disables usage checking. The default is to perform alias checking. When alias checking is enabled, the compiler may insert run-time alias checks, and can often generate more efficient code. Details of alias and usage checking rules are given in the appendices of the '*occam 2 Toolet Language and Libraries Reference Manual*' and also in the '*occam 2 Reference Manual*'.
B	Displays messages in brief (single line) format.
C	Disables the generation of object code. The compiler performs syntax, semantic, alias and usage checking only.
CODE *nnn*	Specifies how large to make the code buffer. If not specified, the compiler will allocate 240 Kbytes. The code buffer is expressed as Kbytes, e.g. to allocate a buffer = 100kbytes, specify CODE 100.
D	Generates minimal debugging information. The default is to produce full debugging information. Debugging data is required by the debugger.
E	Disables the use of the compiler libraries. This prevents the compilation of some programs which require 'complicated' arithmetic such as real arithmetic on a processor which does not have a floating point unit. If this option is used and the occam code requires use of the libraries, an error is reported.
G	Enables the compiler to recognize the restricted range of transputer instructions via the ASM construct, as listed in the 'Transputer code insertion' appendix of the '*occam 2 Toolset User Guide*'.
H	Produces code in HALT mode. This is the default compilation mode and may be omitted for HALT mode programs. (See options S and X also).
HELP	Displays a full help page.
I	Displays additional information as the compiler runs. This information includes target and error mode, and information about directives as they are processed. The default is not to display this information.
K	Disables run-time range checking. The default is to insert run-time range checks. See section 3.6.
N	Disables usage checking. The default is to perform usage checking. Usage checking is also disabled by option 'A'. When usage checking is enabled the compiler is able to generate more efficient code. Details of alias and usage checking rules are given in the appendices of the '*occam 2 Toolset Language and Libraries Reference Manual*' and also in the '*occam 2 Reference Manual*'.

Option	Description
NA	Disables the insertion of run-time checks for calls to ASSERT.
NWCA	Disables warnings when CHAN OF ANY is used. See section 3.7.
NWGY	Disables warnings when the obsolete construct GUY is used. See section 3.7.
NWP	Disables warnings when function or procedure parameters are not used. See section 3.7.
NWU	Disables warnings when declared variables or routines are not used. See section 3.7.
O *outputfile*	Specifies the name of the output file. If no output file is specified the compiler uses the current directory and input filename and adds a .tco extension.
P *filename*	Generates a map file giving details of code mapping in memory. A filename must be specified. Map files can be displayed as normal text files and are read by the imap tool, see section 3.12. **Note:** if this file is to be used as input to imap, it must be given an extension of the form: .mxx. The characters 'xx' are determined by the 2nd and 3rd characters of the extension given to the compiler object file. For example if the compiler object file takes the default extension .tco, the information file is given the extension .mco.
QS	Generates code which favours space efficiency. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate more compact code.
QT	Generates code which favours faster execution times. This is the default. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate the faster code.
R *filename*	Redirects error messages to a file.
S	Produces code in STOP mode. (See options H and X also).
U	Disables the insertion of code to perform run-time error checks. The default is to perform run-time error checks. See section 3.6.
V	Prevents the compiler from producing code which has a separate vector space requirement. The default is to produce code which uses separate vector space. See the appendix in the '*occam 2 Toolset Language and Libraries Reference Manual*' which describes occam 2 implementation details.
W	Enables the compiler to recognize the full range of transputer instructions via the ASM construct.
WALIGN	Warns whenever a runtime alignment check is inserted for a RETYPE. See section 3.7.
WALL	Enable all warnings which are controlled from the command line. See section 3.7.
WD	Provides a warning whenever a name is descoped. See section 3.7.
WO	Provides a warning whenever a run-time alias check is generated. See section 3.7.
WQUAL	Enables software quality warnings, see section 3.7.
X	Produces code in UNIVERSAL mode. See section 3.5. (See options H and S also).
Y	Disables the use of library calls for channel input and output and instead uses transputer instructions. See section 3.9. **Note:** This option is incompatible with the 'software virtual routing' facilities of the configurer.

Table 3.1 occam 2 compiler options

3.2.1 Default command line arguments

Commonly used command line parameters can be defined in the host environment variable OCARG. Parameters specified in this way are automatically added to the start of the command line when the compiler is invoked.

Command line parameters must be specified in OCARG using the syntax required by the oc command line.

3.3 Filenames

occam source files can be given any legal filename for the host system you are using. The use of the .occ extension for occam source, and the .inc extension for files containing declarations of constants and protocols, is recommended. If an extension is not specified for the input file, the compiler will assume the extension is .occ.

Output files are specified using the 'o' option. If you do not specify a filename, the input filename is used (minus any directory name) and a .tco file extension is added. In this case the file will be placed in the current directory i.e. the directory from which the compiler is invoked.

If you use the Makefile generator tool **imakef** you *must* use the extensions described in section 12.3.

3.4 Transputer targets

The compiler generates code for a specific transputer type. This means that a processor type should be specified for all transputer targets. If more than one processor type is specified, the compilation will terminate immediately and an error message will be displayed.

Transputers are also grouped into classes for the purpose of generating common code suitable for running on a number of different transputer targets. Transputer classes group transputers according to word size and instruction set compatibility. They can be used to generate code for combinations of transputers.

The use of transputer types and classes in developing programs is explained in appendix B. The command line options for selecting a transputer target are also given in this appendix.

3.5 Error modes

The execution error mode determines the behavior of a program if it fails during execution. There are two main modes; HALT system and STOP process. There is also a special mode called UNIVERSAL. Command line options are provided to select the error mode for the compilation. Specifying more than one error mode will cause the compilation to terminate immediately and an error message will be displayed. The execution behavior of programs compiled in the different modes is as follows:

HALT When an error occurs in the program the transputer halts. This is useful for developing and debugging systems and is the default mode. For errors to be detected correctly the server must be invoked with the 'SE' option.

STOP When an error occurs the system behaves like the occam STOP process, that is the process causing an error does not continue. Other processes continue until they become dependent upon the stopped process. This ensures that a failure in one process does not automatically produce failure in other processes. Using this mode it is possible to build a system with redundancy and enable a system to run even if parts of the program fail or processes fail because a time out is exceeded.

UNIVERSAL UNIVERSAL mode enables the user to compile code that may be run with either HALT or STOP mode in effect. The decision about which mode to adopt need not be taken until the separately compiled modules are combined into a linked object file. On linking the modules, any code that has been compiled in UNIVERSAL error mode will adopt the error mode of the other modules i.e. either HALT mode or STOP mode. HALT and STOP error modes may not be combined on the same processor.

Code compiled in either HALT or STOP mode may call code compiled in UNIVERSAL mode, however, code compiled in UNIVERSAL mode may only call code which has also been compiled in UNIVERSAL mode. It cannot call code which has been compiled in HALT or STOP mode.

All separately compiled units for a single processor must be compiled for compatible error modes. Where a library is used the module with the appropriate error mode will be selected.

Note: The implementation of error modes on the T9000 is done via a trap handler (installed automatically by the configurer), which gives either HALT or STOP behavior when an error occurs.

Compilation error modes and their effects are described in more detail in the '*Programming single transputers*' chapter of the '*occam 2 Toolset User Guide*'.

3.6 Enable/Disable Error Detection

By default the compiler inserts code to execute run-time checks for errors it cannot detect at compile time. In some circumstances it may be desirable to omit the run time error checking in one part of a program, for example, in a time-critical section of code, while retaining error checks in other parts of a program, for debugging purposes. Three command line options are provided to enable the user to control the degree of run-time error detection performed; they are the 'K', 'U' and 'NA' options.

The compiler option 'K' disables the run-time range checks for the module being compiled. Range checking only includes checks on array subscripting and array lengths.

The compiler option 'U' prevents the compiler from inserting any code to explicitly perform run-time checks. This will disable run-time checks associated with type conversion, shift operations, array access, range validation and replicated constructs such as SEQ, PAR, IF, and ALT. Runtime checks implicit in the transputer instructions are still performed, for example, *add* will automatically check for arithmetic overflow.

Note: The 'U' option can be used to remove unnecessary runtime checks from code which is fully debugged and known to be error-free. It is equivalent to implementing the occam error mode UNDEFINED.

The 'NA' option prevents the compiler from inserting any code to check calls to ASSERT. In effect, each ASSERT behaves like SKIP. Any calls to ASSERT which can be evaluated at compile time will still be checked.

The effect of using these options is described in detail in the '*Programming single transputers*' of the '*occam 2 Toolset User Guide*'.

3.7 Enabling/disabling warning messages

There are several command line options which allow the user to either enable or disable the generation of certain warning messages produced by the compiler:

- The NWCA option disables the generation of warning messages when CHAN OF ANY is used. CHAN OF ANY is now considered obsolete and replacement with named protocols of type ANY is recommended. See section '*Language extensions*' appendix of the '*occam 2 Toolset Languages and Libraries Reference Manual*'.

- The NWGY option disables the generation of warning messages when the GUY construct is used. GUY is now considered obsolete and ASM should be used instead.

- The NWP option disables warning messages being generated when parameters to procedures are not used.

- The NWU option disables warning messages being generated when variables or routines are not used.

- The WALIGN option provides a warning whenever a runtime alignment check is inserted for a RETYPE.

- The WALL option turns on all warnings which are controlled from the command line i.e. it is currently equivalent to WD, WO, WALIGN and WQUAL.

- The WD option provides a warning whenever a name is descoped, for example when a name is used twice and one occurrence of it is hidden within an inner procedure. See section 8 of the '*occam 2 Reference Manual*' for details of occam scope rules.

- The WO option provides a warning whenever a run-time alias check is generated i.e. to check that variables do not overlap. These checks generate extra code and the user may wish to be alerted to this.
- The WQUAL option enables software quality warnings. Currently these include a warning about incorrect positioning of PLACE statements and a warning about unused CASE options.

Section 3.14.1 lists the warning messages which are affected by these options.

3.8 Support for debugging

The occam 2 compiler supports interactive debugging with the debugger by default.

The compiler 'D' option disables the generation of full debugging data. Minimal debugging data remains to allow the debugger to backtrace through the code. This option enables library code to be created without the overhead of debugging data.

3.9 Channel input/output

By default the compiler will generate calls to library routines to perform channel input and output, rather than using the transputer's instructions. The compiler's 'Y' option forces the compiler to use sequences of transputer instructions for channel input and output, resulting in faster code execution.

Note: that code which is compiled to use transputer instructions for channel input/output may call code which uses library routines to perform channel input/output, but *not* vice versa.

Note: The 'Y' option is incompatible with the 'software virtual routing' facilities of the configurer. When software processes are used to implement the routing of channel communications between non-adjacent transputers, channel input/output must be implemented by library calls. See the chapter about 'Configuration' in the '*occam 2 Toolset User Guide*' for further details.

3.10 Separately compiled units and libraries

Any group of one or more occam procedures and/or functions may be compiled separately provided they are completely self-contained and make no external references except via their parameters or compiler directives. Separate compilation is used to reduce the need for recompilation, and to split compilations into smaller parts. Separately compiled code is known as a compilation unit.

Any collection of compilation units may be made into a library using the librarian `ilibr` (see chapter 9). Libraries and compilation units differ in the following way:

- Libraries are selectively loaded as required by the transputer type and error mode of the compilation, whereas separately compiled units are *always* loaded. If a unit containing incompatible code is used an error is generated, whereas libraries containing incompatible code are ignored.

All separate compilation units and libraries must be compiled before the program that references them is itself compiled. An easy way to ensure this is to use the toolset Makefile generator `imakef` with a suitable Make utility. For more details see Chapter 12.

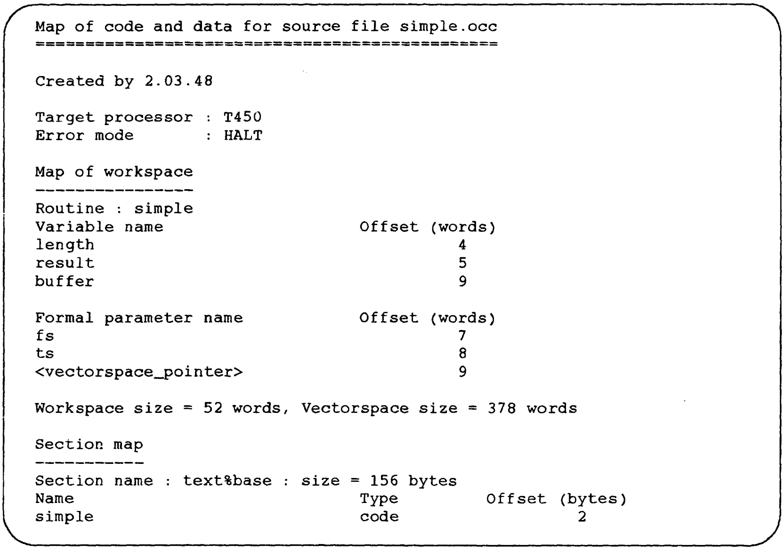
3.11 Code insertion using ASM

Two compiler options are provided to enable the compiler to recognize transputer instructions inserted into source code using the ASM construct. The 'G' option permits use of a limited range of sequential instructions whereas the 'W' option permits use of the full range of transputer instructions. For further details see the '*Transputer Code Insertion*' appendix of the '*occam 2 Toolset User Guide*'.

3.12 Memory map

The compiler may be instructed, via the P *mapfile* option, to produce a map of workspace for each function defined in the file. The file contains information which may assist the user during program debugging and can be used as input to the memory mapper `imap`. The map is written to the file *mapfile*.

The file consists of a series of workspace maps, one for each routine, giving details of workspace requirements. These are followed by a section map listing details of procedures and functions.



```
Map of code and data for source file simple.occ
===============================================

Created by 2.03.48

Target processor : T450
Error mode       : HALT

Map of workspace
----------------
Routine : simple
Variable name                     Offset (words)
length                                     4
result                                     5
buffer                                     9

Formal parameter name             Offset (words)
fs                                         7
ts                                         8
<vectorspace_pointer>                      9

Workspace size = 52 words, Vectorspace size = 378 words

Section map
-----------
Section name : text%base : size = 156 bytes
Name                              Type         Offset (bytes)
simple                            code                 2
```

Figure 3.1 Example compiler map

The file is generated in text format. The following information is present:

- The name of the source file for which the map of code and data is being produced.
- Version data for the compiler.
- The target transputer of the compilation, T450, T805, T400, etc.
- The error mode of the compilation.
- Name of the routine for which the map of workspace is being produced. Items in the workspace map are given in ascending order of workspace offset.
  - List of local variables giving their offset (in words) into the routine's workspace. This list may include temporary variables introduced by the compiler.
  - List of formal parameters giving their name and offset (in words) into the routine's workspace. Parameters added by the compiler may also be listed, see Table 3.2 for a list of their names. Further details of these parameters can be found in the appendix of the '*occam 2 Toolset Language and Libraries Manual*' which describes occam 2 implementation details.
  - The workspace and vector space requirements of the routine in words. This includes the requirements of all nested calls but *not* the four word overhead introduced by the transputer call instruction.
- Name of the section for which the section map is being produced. Items in the section map are given in ascending order of section offset.

Details of how the compiler allocates space for variables are given in the appendix of the '*occam 2 Toolset Language and Libraries Manual*' which describes occam 2 implementation details.

Formal parameter
hidden dimension
vectorspace pointer
static link

Table 3.2 Parameters inserted by compiler

Note: The message "`No local variables`" may be displayed if no user variables are found, however, compiler temporaries may have been assigned to workspace. In addition some compiler temporaries may not be listed in the map file.

Information generated in the compiler map file may be extracted by the `imap` tool. This tool can be used to produce a memory map for the program after it has been compiled, linked and collected. See chapter 13.

3.13 Compiler directives

The occam compiler supports a number of directives that allow the programmer to customize a compilation. All are extensions to the language. If the compiler '`I`' option is used directives are displayed on the screen as the compilation proceeds.

Directives supported are:

`#INCLUDE` – inserts occam source code

`#USE` – references separately compiled units and libraries

`#IMPORT` – references non-occam compiled code

`#COMMENT` – inserts comments in object code

`#OPTION` – allows selection of compiler options from within source text

`#PRAGMA` – miscellaneous extensions, including support for the import of other languages, code placement in RAM, and disablement of checks on specific variables.

3.13.1 Syntax of compiler directives

Filenames referred to in compiler directives must be enclosed in double quotes ("). Files are located according to the search strategy defined in section A.4.

If double quotes are to be used within a directive, the double quote character must be preceded by an asterisk (*).

The scope of directives are defined, like declarations of constants and protocols, by the level of indentation in the occam source.

When `imakef` is used, if a filename in a `#USE`, `#INCLUDE` or `#IMPORT` directive does not already have an extension then `imakef` will add the appropriate extension depending upon the target that it is attempting to build. If you use the Makefile generator tool `imakef` you *must* use the extensions described in sections 12.3 and A.6.

3.13.2 `#INCLUDE`

The `#INCLUDE` directive inserts the contents of a named file at the point in the program source where the directive occurs, with the same indentation as the directive.

`#INCLUDE` files can be used by any number of programs, including separately compiled units, and are commonly used to share common declarations of constants and protocols between several programs.

To track file dependencies within included files use of the `imakef` tool is recommended.

The syntax of the `#INCLUDE` directive is as follows:

`#INCLUDE` *"filename" [comment]*

where: *filename* is the name of the file to be included. The extension must be supplied.

comment is any text preceded by the characters '—'.

The first text after the directive must be the filename enclosed within double quotes (").

All other text on the line is ignored and may be used for comments. For example:

```
#INCLUDE "header.inc"
```

Included files may be nested to any depth.

3.13.3 #USE

The #USE directive allows separately compiled occam units and libraries (in TCOFF format) to be referenced from occam source. The file referenced by the #USE directive must be compiled for a compatible processor type and compilation mode as the main program, and should be made available in all modes for which the program will be compiled.

The compiler ignores all library modules compiled with a processor type or compilation mode incompatible with the current compilation. A library may be used in any number of separately compiled units or other libraries, provided that each unit contains the #USE directive.

Note: that if a library, which will be read by an occam compiler #USE directive, contains routines of the same name, these routines *must* have the same interface i.e. result type and parameter list, and should have the same functionality.

Any names in the library which do not conform to occam syntax, and which have not been translated by means of a TRANSLATE pragma will be ignored. **Note:** this means that a TRANSLATE pragma must precede its related #USE directive. See section 3.13.7.

The syntax of the #USE directive is as follows:

#USE *"filename" [comment]*

where: *filename* is the name of the object code file. The object file can be a compiled (.tco) or library (.lib) file. If you omit the file extension, the compiler adds the extension of the output file. This will be .tco unless you specified an output filename using the 'o' option.

comment is any text preceded by the characters '—'.

The first text after the #USE directive must be the filename, which must be enclosed within double quotes ("). All other text on the line is ignored and may be used for comments. For example:

```
#USE "module"
#USE "library.lib"
#USE "module.tco"
#USE "module.t2h"
```

3.13.4 #IMPORT

The #IMPORT directive allows code produced by compatible non-occam compilers to be referenced from occam programs. It operates in the same way as #USE except that the code that is imported is marked as 'foreign' and not included in makefile searches.

Note: The code imported by `#IMPORT` must be compatible with the current toolset linker `ilink`.

The syntax of the `#IMPORT` directive is as follows:

`#IMPORT` *"filename" [comment]*

where: *filename* is the name of the compiled equivalent occam process. If no extension is given the `.tco` extension is assumed.

comment is any text preceded by the characters '—'.

The first text after the `#IMPORT` directive must be the file name, which must be enclosed within double quotes ("). All other text on the line is ignored and may be used for comments.

An example of how to use the `#IMPORT` directive is given below:

```
#IMPORT "centry.lib"  — C interface code
          .
          .
          .

PROC.ENTRY(fs, ts, flag, wsl, ws2, in, out)
  — call C language program
```

The parameters supplied in the program call, `flag`, `wsl`, `ws2`, `in`, and `out` are those of the type 2 procedural interface. The program must be linked with C libraries `centry.lib` and `libc.lib`. Details of this method of mixed language programming can be found in the '*occam 2 Toolset User Guide*'.

Details of an alternative method of mixed language programming where non-occam programs are called directly using library functions can also be found in the '*Mixed language programming*' chapter of the '*occam 2 Toolset User Guide*'.

3.13.5 #COMMENT

The `#COMMENT` directive allows comments to be placed in the object code. These comments can be read by the binary lister tool `ilist`.

The syntax of the `#COMMENT` directive is as follows:

`#COMMENT` *"string" [comment]*

where: *string* is the text of the comment. Comments must be enclosed in double quotes following the `#COMMENT` directive. Comments cannot be split over more than one line.

Comments may not appear at the exact position in the object code corresponding with the source code directive, but the sequence of comments in the file is always maintained. Comments included by #COMMMENT are stripped from the object code when it is linked or made bootable.

The main use for the #COMMENT directive is in libraries or other pre-compiled code where it can be used to indicate a version number, record dependencies on other libraries, and hold copyright information. The comment strings can then be displayed using ilist.

An example of how to use the #COMMENT directive is given below:

```
PROC my.lib ()

  #COMMENT "My library V1.3, 18 March 1995"
  #COMMENT "Copyright me 1995"

  SEQ
    ... library source
```

#COMMENT acts like #PRAGMA COMMENT. For example:

```
#COMMENT "string"
```

is equivalent to:

```
#PRAGMA COMMENT "string"
```

See also section 3.13.7.

3.13.6 #OPTION

The #OPTION directive allows you to specify certain command line options within the source text of a compilation unit, so that they apply only to that unit. Options specified in this way are simply added to the command line when the compiler is invoked.

Arguments to #OPTION are those that relate directly to the source, namely:

A – disable alias (and usage) checking.

E – disable the compiler libraries.

G – allow sequential code inserts (via the ASM construct).

K – disable the insertion of run-time range checks.

N – disable usage checking.

U – disable the insertion of any run-time error checks.

V – disable separate vector space usage.

W – enable full code inserts, (via the ASM construct).

Y – disables channel i/o by library calls, instead transputer instructions are used.

Specifying any other option produces an error. Descriptions of the arguments can be found in Table 3.1.

#OPTION directives can only appear in the file to which they apply; they cannot be nested in an included file. #OPTION directives must also be the first non-blank or non-comment text in the source file. If they are found at any other position in the file an error is reported.

The syntax of the #OPTION directive is as follows:

#OPTION *"optionname {optionname}" [comment]*

where: *optionname* is any option permitted in a #OPTION directive. Spaces within the double quotes are ignored. No option prefix character is required in the syntax and none should be specified.

comment is any text preceded by the characters '—'.

The first text after the #OPTION directive must be the list of options enclosed in double quotes. All other text on the line is ignored and may be used for comments.

An example of how to use the #OPTION directive is given below. In the example the unit does not require usage checking but contains transputer code inserts from the restricted set.

```
— This compilation unit requires sequential
— code inserts and does not pass the usage check.

#OPTION "G N"

PROC x ()
  ...  body of procedure
:
```

The #OPTION directive should only be used for compiler options that are *always* required for a specific compilation.

UNDEFINED error mode

#OPTION "U" can be used to implement the occam error mode UNDEFINED.

3.13.7 #PRAGMA

The #PRAGMA directive is provided to reference segments of code for mixed language compilations and/or linking functions:

#PRAGMA *pragma-name {optional values} [comment]*

where: *pragma-name* may be one of:

```
COMMENT
EXTERNAL
LINKAGE
PERMITALIASES
SHARED
TRANSLATE
```

optional values may be specified for each type of pragma. The values that the options may take are specific to the pragma being used; they are described below.

comment is any text preceded by the characters '—'. All pragma types may have a comment appended to them.

`#PRAGMA COMMENT` *"string" [comment]*

`#PRAGMA COMMENT` allows comments to be placed in the object code. These comments can be read by the binary lister tool `ilist`. In all respects `#PRAGMA COMMENT` acts like `#COMMENT` (see section 3.13.5).

`#PRAGMA EXTERNAL` *"declaration" [comment]*

This directive allows access to other language compilations. *declaration* is a `PROC` or `FUNCTION` declaration, with formal parameters which correspond to the required calling convention. This is followed (within the string) by two numbers in decimal, indicating the number of workspace slots (words) and optionally the number of vectorspace slots to reserve for that call. The number of vectorspace slots defaults to 0. The number of the workspace slots should not include those needed to set up the parameters for the call. **Note**: that if the vectorspace requirement is zero, then no vectorspace pointer parameter will be passed to the routine.

It is important to ensure that enough space is allocated, both for workspace and vectorspace, because the compiler cannot check for overruns.

The syntax of the declaration is as follows:

formal procedure or function declaration = workspace [, vectorspace]

Examples:

```
#PRAGMA EXTERNAL "PROC p1 (VAL INT x, y) = 20"
#PRAGMA EXTERNAL "PROC p2 (VAL INT x, y) = 20, 100"
#PRAGMA EXTERNAL "INT FUNCTION f1 (VAL INT x, y) = 50"
#PRAGMA EXTERNAL "INT FUNCTION f2 (VAL INT x, y) = 50, 0"
```

The procedure or function name is the name by which the external routine is accessed from the occam source. It is also the name which will be used by the linker to access the external language function, though this may be modified by use of the `TRANSLATE` pragma.

`#PRAGMA LINKAGE` *["section-name"] [comment]*

This pragma enables the user to identify modules that he wishes to be placed in on-chip RAM. The user may then prioritize the order in which these modules are linked together by using a linker directive. On-chip RAM is allocated to workspace first and then to code. Provided there is enough RAM available it should be possible for commonly used subroutines to be processed in the on-chip RAM. This should make the program run faster.

Normally the compiler creates the object code in a section named "text%base". The #PRAGMA LINKAGE directive causes the compiler to change the name of the section to that supplied in the string. If the directive is used but no section name is provided by the user, the compiler supplies the priority section name "pri%text%base". More than one module may take the section name "pri%text%base".

A linker directive is used to change the order in which code modules are linked together, by supplying a list of prioritized *section-names*, see section 10.4.6. Provided that the linker does not encounter any linker directives listing *section-names*, it will place "pri%text%base" modules first.

Note: floating point routines such as REAL32OP and REAL32OPERR are automatically optimized by the compiler by placing them in a "pri%text%base" section.

The #PRAGMA LINKAGE directive should appear at the start of the source code, immediately following the #OPTION directive, if one is present.

For example:

```
#OPTION "N"
#PRAGMA LINKAGE "PRIORITY1" – highest priority
```

#PRAGMA PERMITALIASES *variable.list [comment]*

This pragma disables alias checking for specified variables. It may be applied to normal variables, abbreviations and RETYPES, or non-VAL formal parameters.

See the '*occam 2 Toolset Language and Libraries Reference Manual*' for a description of alias checking.

This pragma must immediately follow the declaration of the variables to which it refers. For example:

```
INT x, y, z :
#PRAGMA PERMITALIASES x .....
#PRAGMA PERMITALIASES y .....        is correct

INT x :
INT y :
#PRAGMA PERMITALIASES x .....        is incorrect
```

If the pragma refers to formal parameters, it must immediately follow the procedure heading.

To allow the compiler to generate efficient code, it is preferable to use the #PRAGMA PERMITALIASES on individual variables, rather than applying the 'A' command line option on the whole compilation. However, the most efficient code will be generated when neither the PERMITALIASES pragma nor the 'A' command line option are used.

#PRAGMA SHARED *variable.list [comment]*

This pragma disables usage checking for specified variables. It may be applied to normal variables, abbreviations and RETYPES, or non-VAL formal parameters.

See the '*occam 2 Toolset Language and Libraries Reference Manual*' for a description of usage checking.

This pragma must immediately follow the declaration of the variables to which it refers. If these are formal parameters, it must immediately follow the procedure heading. See pragma PERMITALIASES above.

To allow the compiler to generate efficient code, it is preferable to use the #PRAGMA SHARED on individual variables, rather than applying the 'N' command line option on the whole compilation. However, the most efficient code will be generated when neither the SHARED pragma nor the 'N' command line option are used.

#PRAGMA TRANSLATE *identifier "string"* [*comment*]

This is used to enable linkage with routines whose entry point names do not correspond to occam syntax for identifier names; both imported names to be called by this compilation unit and exported names defined in this compilation unit. An entry point is a name which is visible to the linker. Thus procedures and functions declared at the outermost level of a compilation unit are entry points, whereas nested procedures and functions are not.

Any entry point defined in the compilation unit whose name matches *identifier* is translated to *string* when inserted into the object file, and hence can only be referenced as *string* when linking.

String may not contain the following characters: the NUL character ('*#00'), space(' '), or open and close parentheses ('(' and ')').

Any entry points in #USEd libraries and other compilation units whose names match *string* can be referred to within the compilation unit as *identifier*. This also applies to *identifiers* defined by EXTERNAL pragmas. TRANSLATE pragmas must precede any reference to their identifier.

For example:

```
#PRAGMA TRANSLATE c.routine "c_routine"
#PRAGMA EXTERNAL "PROC c.routine () = 100"
```

3.14 Error messages

All messages produced by the compiler are in the standard toolset format. Details of the format can be found in section A.7. Messages are generated at severity levels *Information, Warning, Error, Serious* and *Fatal.*

No object files are generated if a message occurs at severity levels *Error, Serious* or *Fatal.*

Notes

1 The compiler libraries are automatically loaded if required, unless the compiler 'E' option is used.

2 The compiler finds the compiler libraries by searching the path specified by the host environment variable ISEARCH. The most common cause of a compiler library error is failure to set up this environmental variable correctly.

The error messages listed here are those which are produced by incorrect use of the compiler, caused for instance by failing to specify command line options correctly. The compiler also reports all syntax and semantic errors found in the program; these messages are not listed here as they are language specific and therefore outside the scope of this document.

3.14.1 Warnings

Badly formed #PRAGMA *name* **directive**

The pragma directive does not conform to the required syntax.

CHAN OF ANY is obsolete: use PROTOCOL name IS ANY

The `CHAN OF ANY` construct is now considered obsolete. The ability to define a named protocol as in `PROTOCOL name IS ANY` provides greater security and should be used in preference. This warning may be disabled by means of the `NWCA` command line switch.

GUY construct is obsolete: use ASM instead

The `GUY` construct is obsolete; the `ASM` construct provides greater security and should be used in preference. This warning message may be disabled by means of the `NWGY` command line switch.

name **is not used**

The named variable is never used. This warning may be disabled by the `NWU` command line option.

name **placed below MEMSTART**

The named variable has been placed below **MemStart**.

name **placed below MEMSTART**

The named variable has been placed below **MemStart** on one of the byte-mode link control words.

Illegal use of PROTOCOL tag name in an expression

A `PROTOCOL` tag name has been used in an expression. This is illegal and will not be permitted in future compilers.

Name *name* **descopes a previous declaration**

This name descopes another name which has already been declared. This warning is only enabled when the `WD` command line option is used.

No compatible entrypoints found in *name*

The named library contains no routines which may be called from this error mode and/or processor type.

Obsolete channel type conversion: use channel RETYPE

The ability to pass a CHAN OF ANY as an actual parameter to a procedure whose formal parameter is a different channel type is obsolete. A channel RETYPE should be inserted before the call to make the type conversion explicit. This warning may be disabled by means of the NWCA command line switch.

Parameter *name* is not used

The named parameter is never used. This warning may be disabled by the NWP command line option.

Placement expression for *name* clashes with virtual routing system

The named variable is placed on one of the transputer links. This may interfere with the interactive debugging system or the virtual routing system.

Placement expression for *name* wraps around memory

The calculation of the machine address for this variable has overflowed; the truncated address is used.

Possible side-effect: PLACED variable *name*

A PLACEd variable has been declared inside a VALOF. The compiler cannot ensure that this cannot cause a side-effect.

Possible side–effect: instanced PROC has PLACED variable name

A PLACEd variable has been declared inside a PROC which is called from within a VALOF. The compiler cannot ensure that this cannot cause a side-effect.

PORT *name* must be placed

A PORT type must be placed using an allocation. See the '*occam 2 Reference Manual*' for further details.

PRAGMA or PLACEment must immediately follow declaration of *name*

There should be no other variable declarations between a variable's own declaration and any PLACE statement.

Routine *name* is not used

The named routine was never called. This warning may be disabled by the NWU command line option.

Run-time allignment check required for RETYPE

A RETYPE is being used to convert from one type to another with a more restrictive alignment requirement. The compiler inserts a run-time check to ensure that the object is suitable aligned. This warning is only enabled by means of the WALIGN command line option.

Run-time disjointness check inserted

number **Run-time disjointness check inserted**

The compiler has inserted run-time checks to ensure that variables are not aliased (i.e. that they do not overlap). This warning is only enabled when the WO command line option is used.

Tag *name* is not handled in CASE input

Tag *name* appears in the channel's PROTOCOL, but no guard for it appears in this CASE input. This warning is only enabled by means of the WQUAL command line switch.

TRANSLATE ignored: Module containing *name* has already been loaded

The #TRANSLATE pragma must precede any #USE of a library containing that string.

TRANSLATE ignored: Name *name* has already been used

You may not specify multiple translation strings for the same name.

TRANSLATE ignored: String contains NUL character

The specified string for a #TRANSLATE pragma may not include a NUL (zero) byte.

TRANSLATE ignored: String *name* has already been used

You may not specify multiple names to be translated to the same string.

Unknown #PRAGMA name: *name*

The pragma name is ignored.

Using length '*name*' in array part of counted array input is obsolete

The language no longer permits using the length part of a counted array input to appear in the array part. It does however allow the following special case to be written where the length only appears as the length of a slice:

channel.exp ? *name* :: [*array.exp* FROM 0 FOR *name*]

This is transformed by the compiler into the equivalent construct:

channel.exp ? *name* :: *array.exp*

The former construct is obsolescent and programs should be re-written to use the latter form.

Workspace clashes with variable PLACED AT WORKSPACE *number*

A variable has been placed at the workspace address *number*, and this clashes either with another placed variable, or with the compiler's workspace allocation requirements.

3.14.2 Errors

Bad object file format

Library or separately compiled procedure object code is not in the correct format. The code may not have been linked correctly, or the file may have become corrupted.

Badly formed compiler directive

A compiler directive following # was not recognized.

Badly formed #EXTERNAL directive

The number of workspace slots to reserve for the call has not been specified or negative workspace or vector space slots have been specified in error.

Cannot open file "*string*"

File is missing, or file system error.

Cannot open output file

The object file could not be opened. File system error.

Cannot open output file (*string*)

The file given as parameter to the command line R option could not be opened.

Cannot open source file

The source file cannot be opened. Either it does not exist, or there is a file system error.

Code buffer full (*nnn* bytes); use command line to increase buffer size

The compiler has an internal buffer for code which is about to be placed into the object file; this has overflowed. The CODE command line option may be used to increase the size of this buffer.

Code insertion is not enabled

You must use the G or W options to enable assembler inserts.

Descriptor has incorrect format

Library or separately compiled procedure object code is not in the correct format. The code may not have been linked correctly, or the file may have become corrupted.

Duplicate error modes on command line

Multiple error modes may not be specified for the compilation.

Duplicate processor types on command line

Multiple processor types may not be specified for the compilation.

Expected string after #COMMENT

`#COMMENT` directive must be followed by a string containing the comment.

Expected string after #OPTION

`#OPTION` directive must be followed by a string containing the options .

'*Filename*' **is not a valid object file**

Library or separately compiled unit object code is not in the correct format. The code may not have been linked correctly, or the file may have become corrupted.

Instruction is not available in current code insertion mode

You must set the `W` option in order to use this instruction.

Instruction is not available on target processor

The given instruction is not present in the target instruction set.

Invalid command line option (*string*)

The user specified an unrecognized command line option.

Missing filename

Filename is missing on `#USE`, `#INCLUDE` or `#IMPORT` directive.

Missing object file name

There is no object file name parameter to the command line O option.

Missing output file name

There is no output file name parameter to the command line `R` option.

No filename given

No source file was specified on the command line.

number **reading source file**

File system error. The source file or an include file could not be read. *number* is the host file system error number.

***number* writing to object file**

File system error. The object file could not be written to. *number* is the host file system error number.

Option in illegal position

Only one `#OPTION` directive is allowed in a file, and it must be on the first non-blank or non-comment line in a file.

PRAGMA or PLACEment must immediately follow declaration of *name*

There should be no other variable declarations between a variable's own declaration and any `#PRAGMA`.

Run out of symbol space

The source compilation unit is too large to be compiled.

Unrecognised option "*char*" in option string

Incorrect compiler options specified after a `#OPTION` directive.

4 occonf - occam configurer

This chapter describes the configurer tool `occonf` that configures code for transputer networks. It describes the command line syntax and explains how the tool is used to generate a configuration data file for input to the code collector tool. The chapter ends with a list of error messages.

4.1 Introduction

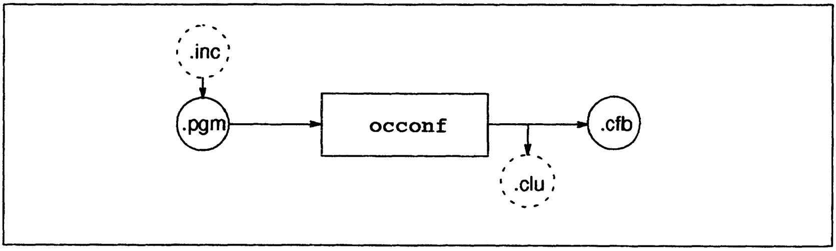
The configurer takes a *configuration description* created using the transputer configuration language and produces a *configuration binary file* which `icollect` uses to generate bootable code for a transputer network. The bootable code may be generated in a specific error mode.

A configuration description describes how code is to be run on a network of transputers. It consists of separate definitions of the software and hardware networks, and a mapping description which defines how the software will be placed on the processor network. Using this description the configurer allocates code to particular processors and performs wide ranging consistency checks on the mapping of software to hardware. The chapter on 'Configuration' in the '*User Guide*' explains how to write a configuration description.

`occonf` enables any topology of software network to be placed on any topology of hardware network. There are no restrictions on how many communication channels may be allocated to a single inter–processor link. Where possible channels should be left unplaced by the user, so that `occonf` can implement the 'best' route through the network.

Linked modules and libraries which are referred to by a configuration description must be already compiled and linked before any file which references them can itself be configured.

The operation of the configurer tool is illustrated below. Files are represented in terms of their standard toolset file extensions.



`occonf` produces two output files:

- the main configuration binary file (`.cfb`) for the collector
- a file (`.clu`) containing executable code packets, used by the collector.

4.2 Running the configurer

To run the configurer use the following command line:

► `occonf` *filename* { *options* }

where: *filename* is the configuration description file. If no file extension is specified, the extension `.pgm` is assumed. Only one file may be specified.

options is a list of one or more options from Table 4.1.

> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Examples of use:

```
oc -t450 simple.occ
ilink -t450 simple.tco hostio.lib -f occam450.lnk
occonf simple.pgm
icollect simple.cfb

oc -t805 simple.occ
ilink -t805 simple.tco hostio.lib -f occama.lnk
occonf simple.pgm
icollect simple.cfb
```

Option	Description
B	Displays messages in brief (single line) format.
C	Disables the generation of object code. The configurer performs syntax, semantic, alias and usage checking only.
CODE *nnn*	Specifies how large to make the code buffer. If not specified, the configurer will allocate 40 Kbytes. *nnn* is a value in Kbytes i.e. the 'K' suffix is not required.
G	Enables the configurer to recognize the restricted range of transputer instructions, via the ASM construct. See section 4.13 and the appendices of the accompanying '*User Guide*'.
GA	Generates a configuration which can be debugged using the *INQUEST* debugger in interactive mode. This option is incompatible with the RO, NV, PRE and PRU options. See section 4.14.
H	Produces code in HALT error mode. This is the default configuration mode and may be omitted for HALT error mode programs. See section 4.7.
HELP	Displays a full help page which lists all the standard options.
I	Displays extra information as the tool runs. This information includes target and error mode, and information about directives as they are processed. The default is not to display this information.
K	Disables run-time range checking. The default is to insert run-time range checking. See section 4.8.
NA	Disables the insertion of run-time checks for calls to ASSERT. See section 4.8.
NV	Generates a configuration without virtual routing. **Note:** any definition of router attributes for processor nodes will be redundant. See section 4.11.
NWCA	Disables warnings when CHAN OF ANY is used. See section 4.12.
NWGY	Disables warnings when the obsolete construct GUY is used. See section 4.12.
NWP	Do not warn if declared parameters are not used.
NWU	Do not warn if declared variables or routines are not used.
O *outputfile*	Specifies an output filename. If no output file is specified the configurer uses the input filename and adds the extension .cfb.
PRE	Generates a configuration which can be profiled by the INQUEST execution profiler. **Note:** This option cannot be used with the **GA** and **PRU** options. See section 4.14.
PRU	Generates a configuration which can be profiled by the INQUEST utilization profiler. **Note:** This option cannot be used with the **GA** and **PRE** options. See section 4.14.
QS	Generates code which favours space efficiency. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate more compact code.
QT	Generates code which favours faster execution times. This is the default. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate the faster code.
R *filename*	Redirects error and information messages to a file.
RA	Creates a file suitable for a boot-from-ROM application in which the user and system processes for the root processor and all other processors are loaded into RAM to execute.

RE	Enables re-ordering of code and data layout in memory, using the `order.code`, `order.vs` and `order.ws` attributes. Also enables the `location.code`, `location.ws`, and `location.vs` attributes. See section 4.9.
RO	Creates a file suitable for a boot-from-ROM application in which the user and system processes for the root processor execute in ROM and for all other processors the user and system processes are loaded into RAM to execute.
S	Produces code in STOP error mode. See section 4.7.
U	Disables the insertion of all extra run-time error checking. The default is to insert run-time error checks. This is a 'stronger' option than K, and can be used to implement the occam UNDEFINED error mode. See section 4.8.
V	Prevents the configurer from producing code which has a separate vector space requirement. The default is to produce code which does use separate vector space.
W	Enables the configurer to recognize the full range of transputer instructions, via the ASM construct. See section 4.13 and the appendices of the accompanying '*User Guide*'.
WALIGN	Warns whenever a runtime alignment check is inserted. See section 4.12.
WALL	Enable all warnings. See section 4.12.
WD	Provides a warning whenever a name is descoped.
WO	Provides a warning whenever a run-time alias check is generated.
WQUAL	Enables software quality warnings, see section 4.12.
X	Produces code in UNIVERSAL error mode. See section 4.7.
Y	The 'Y' option disables the use of library calls for channel input/output and instead uses transputer instructions. See section 4.10.

Table 4.1 `occonf` command line options

4.3 Default command line

Default command line parameters can be defined on the system in the `OCCONFARG` environment variable. Parameters must be specified using the syntax required by the configurer command line.

4.4 Search paths

If a directory path is not specified the configurer uses the standard toolset search mechanism for locating input files, include files, and system library files. Briefly, the current directory is searched first, followed by the directories specified by `ISEARCH` (if defined on the system). For details see section A.4.

4.5 Configurer library files

Depending on the command line options used, the configurer reads a number of special library files which contain system processes. The library files are searched for on the directory specified by ISEARCH. This is normally the toolset libs directory, in which the files were originally installed. The library files are listed in table 4.2.

Library	Description
sysproc.lib	System startup processes for the different transputer types.
sysvlink.lib	Software through-routing processes.
sysdebug.lib	Debugging kernels to support the INQUEST debugger.
sysprof.lib	Profiling kernels to support the INQUEST profiler.

Table 4.2 Configurer library files

4.6 Boot-from-ROM options

The boot-from-ROM options RO and RA indicate that the program is to be collected for loading into EPROM and select the execution mode (from ROM or RAM) for the root transputer code.

4.7 Configuration error modes

The configuration error mode determines the behavior of a program if it fails during execution. The execution behavior of programs configured in the different modes is as follows:

HALT	An error halts the transputer immediately.
STOP	An error stops the process and causes graceful degradation.
UNIVERSAL	Code configured in this mode behave as either HALT or STOP mode, according to the state of the transputer's *HaltOnError* flag.

The error mode selected for the configuration must be compatible with the error mode of the compiled units, referenced by the configuration source. The configurer will produce an error message, if this is not the case.

Table 4.3 indicates the compilation error modes which are compatible and the possible error mode they may be configured for.

Compatible compilation error modes	occonf options
HALT, UNIVERSAL	H
STOP, UNIVERSAL	S
UNIVERSAL	X

Table 4.3 occonf error modes

Compilation error modes and their effects are described in more detail in section 9.3.1. **Note**: that occam UNDEFINED mode can be achieved by using the configurer U option, to disable the insertion of run-time checks. This option behaves in the same way as the U option to the occam compiler, which is documented in the 'Programming single transputers' chapter of the '*User Guide*'.

4.8 Enable/Disable Error Detection

By default the configurer inserts code to execute run-time checks for errors it cannot detect during configuration. In some circumstances it may be desirable to omit the run-time error checking in one part of a program, for example, in a time-critical section of code, while retaining error checks in other parts of a program, for debugging purposes. Three command line options are provided to enable the user to control the degree of run-time error detection performed; they are the `K`, `U` and `NA` options.

The `K` option disables the insertion of run-time range checks on array subscripting and array lengths.

The `U` option prevents the configurer from inserting any code to explicitly perform run-time checks. This option will disable run-time checks associated with type conversion, shift operations, array access, range validation and replicated constructs such as `SEQ`, `PAR`, `IF`, and `ALT`.

The `NA` option prevents the configurer from inserting any code to check calls to `ASSERT`. In effect, each `ASSERT` behaves like `SKIP`. Any calls to `ASSERT` which can be evaluated during configuration will still be checked.

Note: that some checks are still performed; some transputer instructions implicitly check for erroneous conditions.

The `K`, `U` and `NA` options behave in exactly the same way, as the same options provided for the occam compiler. The effects of using these options are described in the 'Programming single transputers' chapter of the '*User Guide*'.

4.9 Enabling memory re-ordering and placement

The `RE` option enables the user to have more control of the layout of code and data areas in memory. When this option is used, the special processor attributes '`order.code`', '`order.vs`' and '`order.ws`' which indicate the relative priority of different data and code areas, and the placement attributes `location.code`, `location.ws`, and `location.vs`, are enabled. See the 'Configuration' chapter in the '*User Guide*' for more details.

4.10 Channel input/output

By default the configurer will generate calls to library routines to perform channel input and output, rather than using the transputer's instructions. The configurer's `Y` and `NV` options, when used together, force the configurer to use sequences of transputer instructions for channel input and output, resulting in faster code execution.

Note: that code which is configured to use transputer instructions for channel input/output may call code which uses library routines to perform channel input/output, but *not* vice versa. This means that an application which includes linked units built with the `Y` option, must be configured with the `Y` and `NV` options to maintain compatibility.

The `NV` option is also described in section 4.11, below.

4.11 Virtual routing

In order to support the virtual routing facilities of the configurer it is required to perform channel input/output using library routines.

The virtual routing facilities of the configurer are normally enabled, that is, the configurer automatically adds virtual routing and multiplexing processes if they are required by the configuration.

If virtual routing is not required the virtual router can be disabled by using the `NV` command line option. In the absence of explicitly supplied multiplexing processes, the normal limit of two channels per link (one in each direction) and a maximum of 4 links applies. If these limits are exceeded and the `NV` option is specified then the configuration will generate errors indicating that channels could not be placed by the configurer. In this case the `Y` or `NV` options should *not* be used at any stage in the application build.

If the `NV` option is to be specified to the configurer then the preceding compilation and linking stages of the application build may be performed using the compiler and linker `Y` option. This is because library i/o will not be required.

If the `NV` option is *not* specified then library i/o must be used for all stages of the application build.

Note: the use of the `NV` option also has an effect on the value of **LoadStart**, see section 4.15.1.

4.12 Enabling/disabling warning messages

There are several command line options which allow the user to either enable or disable the generation of certain warning messages by the configurer:

- The `NWCA` option disables the generation of warning messages when `CHAN OF ANY` is used. `CHAN OF ANY` is now considered obsolete and replacement with named protocols of type `ANY` is recommended. See the *'occam 2 Toolset Language and Libraries Reference Manual'*.
- The `NWGY` option disables the generation of warning messages when the `GUY` construct is used. `GUY` is now considered obsolete and `ASM` should be used instead.
- The `NWP` option disables warning messages being generated when parameters to procedures are declared and not used.
- The `NWU` option disables warning messages being generated when variables or routines are not used.
- The `WALIGN` option provides a warning whenever a runtime alignment check is inserted for a `RETYPE`.

- The WALL option turns on all warnings i.e. it is currently equivalent to WALIGN, WD, WO and WQUAL.
- The WD option provides a warning whenever a name is descoped, for example when a name is used twice and one occurrence of it is hidden within an inner procedure. See section 8 of the '*occam 2 Reference Manual*' for details of occam scope rules.
- The WO option provides a warning whenever a run-time alias check is generated i.e. to check that variables do not overlap. These checks generate extra code and the user may wish to be alerted to this.
- The WQUAL option enables software quality warnings Currently these include warnings for unused options in CASE inputs and warnings about badly positioned PLACE statements.

Section 4.16.1 lists the various warning messages which are affected by these options.

4.13 ASM code

Two configurer options are provided to enable the configurer to recognize transputer instructions, via the ASM construct (see the appendices of the '*User Guide*').

The W option enables the full range of transputer instructions. The G option enables the use of a limited range of sequential instructions. Examples of the use of transputer code insertion can be found in the 'Low level programming' chapter of the '*User Guide*'.

The transputer instruction set is documented in full in the *Transputer Instruction Set – A compiler Writer's Guide.*

4.14 Support for INQUEST

See s.13, book ⑥

Three options are available to support the use of the INQUEST debugger and profiler tools.

The GA option generates a configuration which can be debugged by the INQUEST debugger in interactive mode.

When the GA option is used, the configurer will allocate debugging kernels to all processors which are available for debugging and which have been placed with at least one process. See figure 4.1.

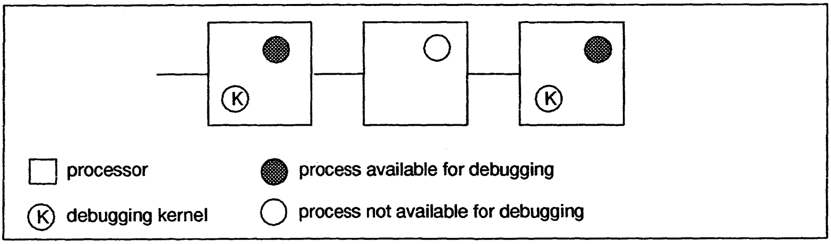

K
K
□ processor
● process available for debugging
Ⓚ debugging kernel
○ process not available for debugging

Figure 4.1 Allocation of debugging kernels using the GA option

Processes may be debugged provided the `nodebug` attribute (set in the `MAPPING` section) is set to `FALSE`, which is the default.

The `GA` option must not be used with the `RO` boot-from-ROM option, or with the `PRE` and `PRU` profiling options.

The `PRE` and `PRU` options generate a configuration which can be profiled by the INQUEST execution and utilization profilers respectively. Processes may be profiled provided the `noprofile` attribute (set in the `MAPPING` section) is set to `FALSE`, which is the default. The `PRE` and `PRU` options are mutually exclusive.

4.15 Default memory map

By default the configurer maps code into memory into the same order as the compiler i.e. beginning at **LoadStart**: workspace; code; separate vector space. The memory segments are contiguous. The upper limit of the memory available to the configurer is defined in the configuration description file (`.pgm` file), by the `memsize` attribute specified for the processor node. The default memory map is illustrated in Figure 4.2.

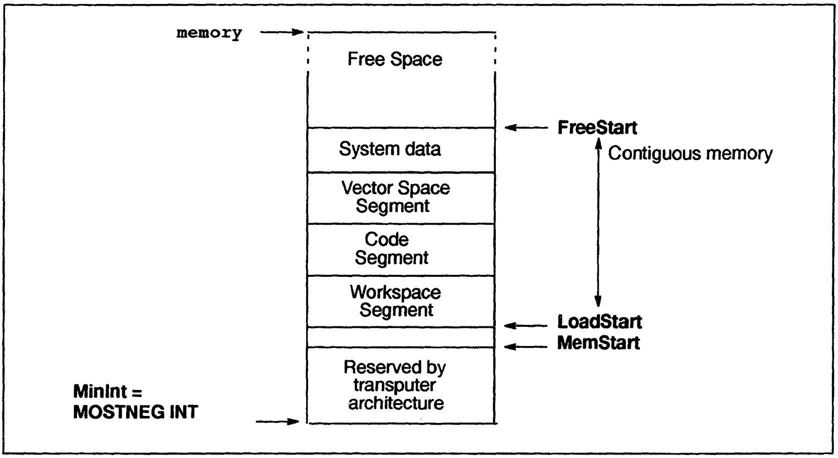


Figure 4.2 `occonf` default memory map

The first 2 or 4 Kbytes of memory (16K for the T450) above `MOSTNEG INT` is implemented as on-chip RAM, and includes a few words which are reserved by the transputer hardware for the implementation of links and other hardware registers. **LoadStart** is either just above or coincident with **MemStart**, see below. **FreeStart** is the start of unused memory.

4.15.1 LoadStart

The position of **LoadStart** for a processor varies depending on the use of `occonf` command line options and the `reserved` processor attribute, optionally specified within a configuration description.

When the `reserved` processor attribute is specified, **LoadStart** is defined to be the memory location obtained by adding the value of `reserved` to `MOSTNEG INT`.

When the `reserved` processor attribute is *not* specified, **LoadStart** is coincident with or just above **MemStart**:

- **LoadStart** = (**MemStart** + 12 words) when the `NV` command line option has *not* been specified i.e. virtual through-routing support is enabled.

  Note: for most processors, **MemStart** is assumed to be greater or equal to 28 words from **MOSTNEG INT**, but if not then **LoadStart** = **MOSTNEG INT** + 40 words. Processors which have a **MemStart** which is less than 28 words from **MOSTNEG INT** are the M212, T212, T222, T225 and T414.

- **LoadStart** = (**MemStart** + 6 words) when the `NV` command line option is specified, disabling virtual through-routing and profiling is enabled.

- **LoadStart** = **MemStart** when the `NV` command line option is specified but profiling is not enabled.

The value of **LoadStart** can be checked once the application has been collected, by generating and examining the collector map file.

4.15.2 System processes

System processes are code and data placed by the configurer for initializing the application. By default, the system startup processes' code and data are placed into user process data areas i.e. workspace or separate vector space. These system processes do not interfere with the user's data because they complete their task before the space is needed by the user's code. **Note:** these system processes do not include the virtual routing processes and profiler and debugger kernels, placed by the configurer.

4.15.3 Configuration description examples

A series of example configuration descriptions are supplied in the `occonf` examples subdirectory. These include configurations for specific network topologies such as rings, grids, trees, and pipelines.

4.16 Configurer diagnostics

If the source code does not conform to the occam 2 configuration language definition, then the configurer will issue diagnostics, in the form of error messages, during the compilation process. When this occurs no object file nor configuration binary file will be produced.

Errors in the configuration source produce diagnostic messages in standard toolset format. Details of the format can be found in section A.7.

Diagnostics are generated at the standard severity levels *Information, Warning, Error,* and *Fatal.* No messages are generated at severity level *Serious.*

Warning messages are listed below.

Note: *if you obtain a configuration diagnostic message which does not appear in any of the following error lists, consult the diagnostic message lists in the* `iconf` *reference chapter. Some messages are common to both* `occonf` *and* `iconf`, *and to avoid duplication are only listed for* `iconf`.

4.16.1 Warning messages

In the following list the **Warning** prefix is omitted for clarity.

Badly formed #PRAGMA ***name*** **directive**

The pragma directive named does not conform to the required syntax.

CHAN OF ANY is obsolete: use PROTOCOL name IS ANY

The `CHAN OF ANY` construct is now considered obsolete. The ability to define a named protocol as in `PROTOCOL name IS ANY` provides greater security and should be used in preference. This warning may be disabled by means of the `NWCA` command line switch.

GUY construct is obsolete: use ASM instead

The `GUY` construct is obsolete; the `ASM` construct provides greater security and should be used in preference. This warning message may be disabled by means of the `NWGY` command line switch.

name **is not used**

The named variable is never used. This warning may be disabled by means of the `NWU` command line option.

Name ***name*** **descopes a previous declaration.**

This name descopes another name which has already been declared. This warning is only enabled by means of the `WD` command line option.

No channel has been placed onto the host connection

`PLACE` or `MAP` statements should be used to define channel input/output across the host connection, if host communication is required.

No direction known for channel ***name*** **on PROCESSOR** ***name***

The configurer cannot determine whether channel *name* is used for input or output on this processor; it will make a guess.

Obsolete channel type conversion: use channel RETYPE

The ability to pass a CHAN OF ANY as an actual parameter to a procedure whose formal parameter is a different channel type is obsolete. A channel RETYPE should be inserted before the call to make the type conversion explicit. This warning may be disabled by means of the NWCA command line switch.

Parameter *name* is not used

The named parameter is never used. This warning may be disabled by means of the NWP command line option.

Placement expression for *name* clashes with virtual routing system

The named variable is placed on one of the transputer links. This may interfere with the INQUEST interactive debugging system or the virtual routing system.

Placement expression for *name* wraps around memory

The calculation of the machine address for this variable has overflowed; the truncated address is used.

Possible side-effect: PLACED variable *name*

A PLACEd variable has been declared inside a VALOF. The compiler cannot ensure that this cannot cause a side-effect.

Possible side-effect: instanced PROC has PLACED variable name

A PLACEd variable has been declared inside a PROC which is called from within a VALOF. The compiler cannot ensure that this cannot cause a side-effect.

Processor *name* unused

The named processor has no code placed onto it.

Routine *name* is not used

The named routine is never called. This warning may be disabled by means of the NWU command line option.

Run-time disjointness check inserted
***number* Run-time disjointness checks inserted**

The configurer has inserted run-time checks to ensure that variables are not aliased (i.e. that they do not overlap). This warning is only enabled by means of the WO command line option.

Unknown #PRAGMA name: *name*

The named pragma is unknown and therefore ignored.

Using length '*name*' in array part of counted array input is obsolete

The language no longer permits using the length part of a counted array input to appear in the array part. It does however allow the following special case to be written where the length only appears as the length of a slice:

channel.exp ? *name* :: [*array.exp* FROM 0 FOR *name*]

This is transformed by the compiler into the equivalent construct:

channel.exp ? *name* :: *array.exp*

The former construct is obsolescent and programs should be re-written to use the latter form.

Workspace clashes with variable PLACED AT WORKSPACE *number*

A variable has been placed at the address *number* in workspace, and this clashes either with another placed variable, or with the configurer's workspace allocation requirements.

4.16.2 Error messages

Most of the messages in this section have been caused by mis–use of the configuration language. For further help see the '*Occam Configuration Language*' appendix in this manual. These messages are generated at the severity level *Error.*

***name* has already been mapped**

name appears twice on the left–hand side of a MAP or PLACE statement.

***name* has already been used as a physical NODE**

Some attributes have already been set for *name*, so it can only be used as a physical node and not as a logical node.

ARC *name* has already been connected

Examine all 'CONNECT' statements creating link connections between processor edges or a processor edge and an external edge.

Attribute *name* has already been set on NODE *name*

Check the declaration of node *name* and any attributes defined for *name* in the mapping and network sections.

Attribute *name* has not been set for NODE *name*

Attribute *name* is a mandatory node attribute and should be set in the NETWORK description.

Attribute *name* may not be SET

You may not SET a link attribute.

Attribute *name* may not be used on logical NODEs

Logical nodes cannot have attributes set for them. Perhaps you meant to set the attribute for a node representing a physical processor.

Attribute *name* set to illegal value on NODE *name*

Check the syntax and range of permitted values for the type of attribute required. For further help see the '*Occam Configuration Language*' appendix in this manual.

CHAN *name* is placed but not properly connected

Check that the ARC that CHAN *name* is PLACEd on is connected within the NETWORK description by a CONNECT statement.

CONNECT expression must be of type EDGE

CONNECT expressions connect two nodes or a node to an external edge by specifying their link connections. They take the form:

'CONNECT *edge* TO *edge* [WITH *arcname*]'

where *arcname* is optional.

Cannot PLACE *name* which was declared outside a PROCESSOR construct

PLACE statement must immediately follow channel declaration.

Cannot disable virtual routing with the INQUEST debugger

You have used the command line options NV and GA together; they are incompatible.

Cannot interactively debug ROM programs

You have used the command line options RO and GA together; they are incompatible.

If you want to debug your program, develop it as a boot-from-link program or boot–from–ROM program with the RA option and debug, and when it is clear of errors re-configure as a boot-from-ROM program with the RO option.

Cannot run both profilers together

You have used the PRE and PRU command line options together, selecting both INQUEST profiler tools. These tools are mutually exclusive.

Cannot run the profiler with the INQUEST debugger

The PRE or PRU command line options have been used in conjunction with the GA command line option; they are incompatible.

Cannot set NODE *name* as root; another has already been set

Examine the other processor declarations, one of them has the `root` attribute set.

Cannot use VAL on a CHAN, PORT, TIMER, or hardware item

Abbreviations of CHAN, PORT, TIMER, NODE, ARC and EDGE cannot use VAL.

Cannot use VAL on a hardware item

It is illegal to use VAL on a NODE, ARC or EDGE.

Channel *name* is mapped onto ARC *name*, not connected to this processor

You need to identify which physical or logical processor is connected to ARC *name*. Either the ARC is connected to the wrong processor or you are trying to map the wrong channel to ARC *name*.

Channel *name* is used for both input and output, but ARC *name* connects to an EDGE

ARC name connects this processor to an edge, so this processor can input on it or output on it but cannot do both. Two channels are required to support two-way communication.

Channel *name* mapped onto unconnected ARC *name*

ARC *name* is required to be connected to a physical link. This is done using a CONNECT statement within the NETWORK description e.g.:

'CONNECT *edge* TO *edge* [WITH *arcname*]'

Channel *name* used for input by more than one process

Channel *name* appears in a previous PROCESSOR statement, defining a process and its channels.

Channel *name* used for output by more than one process

Channel *name* appears in a previous PROCESSOR statement, defining a process and its channels.

Channel *name* with protocol containing INT used across different wordlengths

This is a constraint of the current configurer, see the '*Occam Configuration Language*' appendix in this manual.

Code buffer full (*nnn* bytes); use command line to increase buffer size

The configurer has an internal buffer for code which is about to be placed into the object file; this has overflowed. The CODE command line option may be used to increase the size of this buffer.

EDGE *name* has already been connected

Edge *name* has been connected in a previous CONNECT statement.

EDGE *name* is not connected to rest of network

Edge *name* defines a peripheral device which, if it is to be used, needs to be connected to the rest of the network by a CONNECT statement.

Expected a NODE attribute, found a subscript expression

You appear to have used too many subscripts. The configurer expected to see the name "link" as in:

`CONNECT` *name* `[link] [`*number*`] TO ........`

where *name* is a node name and *number* is the link connection

FUNCTION *name* returns a REAL result but is compiled for wrong calling convention

A routine compiled for class TA can only be run on a T800 if it obeys the correct calling conventions. See the advice for occam code given in section B.2.3. Alternatively re-compile your code for a T8 transputer.

Illegal item in configuration code

An illegal expression has been used in the configuration description. Check the description against the syntax definition, described in the '*Occam Configuration Language*' appendix in this manual.

Illegal item inside CONFIG construct

Check the configuration description against the syntax definition, described in the '*Occam Configuration Language*' appendix in this manual.

Illegal item inside MAPPING construct

Check the configuration description against the syntax definition, described in the '*Occam Configuration Language*' appendix in this manual.

Illegal item inside NETWORK construct

Check the configuration description against the syntax definition, described in the '*Occam Configuration Language*' appendix in this manual.

Illegal processor type "*name*"

Supported processor types are: `"T212" "T222" "T225" "M212" "T400" "T414" "T425" "T800" "T801" "T805" "T450"`

Implementation restriction: Cannot RETYPE INT constants on 16–bit processors; use INT16

This is a constraint of the current configurer, see the '*Occam Configuration Language*' appendix in this manual.

Implementation restriction: Cannot declare *name name* inside a replicator

Declarations of NODEs, ARCs, and EDGEs may not appear inside a replicator.

Implementation restriction: Cannot use INT constant arrays on 16–bit processors; use INT16

This is a constraint of the current configurer, see the '*Occam Configuration Language*' appendix in this manual.

Implementation restriction: Cannot use replicator *name* in channel abbreviation outside a PROCESSOR construct

You have encountered an implementation restriction.

Implementation restriction: INT constant overflows on 16–bit processors; use INT16

This is a constraint of the current configurer, see the '*Occam Configuration Language*' appendix in this manual.

Implementation restriction: root NODE *name* must not be an array element

You have encountered an implementation restriction.

Left hand side of mapping must be of type CHAN

Channels are mapped onto an ARC, with a statement of the following form:

MAP *channel.list* ONTO *arc*

where *channel.list* is a list of channels previously declared to have type CHAN.

Left hand side of mapping must be of type NODE

Logical processors are mapped onto a physical processor, with a statement of the following form:

MAP *processor.list* ONTO *node*

where *processor.list* is a list of logical processors previously declared as NODE types.

Link *name*[link][*number*] has already been connected

Check any previous `CONNECT` statements to identify what

link *name*`[link]`[*number*] is connected to.

Link number *number* is illegal for this NODE

T400 and M212 transputers have just two links, identified `0` and `1`. All other transputers in the T2/T4/T8 series have four links, numbered `0`, `1`, `2` and `3`.

Location attributes ignored because re–ordering isn't enabled

Use the `RE` command line option to enable the `location` attributes.

Multiple CONFIG constructs not permitted

The `CONFIG` keyword introduces the software description, which is a `PAR` or `PLACED PAR` and should include `PROCESSOR` statements defining all required software processes.

Multiple MAPPING constructs not permitted

All `MAP` statements should appear in one section introduced by the `MAPPING` keyword. Any attributes used to control the use of memory or routing should also be included in this section.

Multiple NETWORK constructs not permitted

All statements describing the connectivity and attributes of physical nodes should be placed in one section, introduced by the `NETWORK` keyword.

NODE *name* has not been mapped

Check your mapping description, a logical processor has been declared but not mapped to a physical processor.

No CONFIG construct

The software description is missing or has not been introduced by a `CONFIG` keyword.

No NETWORK construct

The network description is missing or has not been introduced by a `NETWORK` keyword.

No NODE has been specified as root

Configurations which are for boot-from-ROM application must have one physical node defined to be the root transputer. This is performed using the `root` attribute in a `SET` statement.

No hardware route exists from processor *name* for channel *name*

There is no direct link available for channel *name* to use. Because through-routing has been disabled by the `NV` command line option an indirect route cannot be established.

No priority expression permitted when mapping CHANs

The priority expression `PRI` can only be used when mapping logical processors onto physical processors. The priority expression relates to the software process which will run on the processor.

Not enough links from processor *name* for channel *name*

The `NV` command line option has been specified, disabling virtual routing. This means that only one channel can be placed in each direction, onto each link specified in the hardware description.

Ordering attributes ignored because re–ordering isn't enabled

Use the `RE` command line option to enable the `order` attributes.

PRI expression *nn* must evaluate to 0 or 1

Priority expressions are introduced by a `PRI` keyword in a `MAP` statement. `PRI` expressions may take one of two integer values; either 0 indicating *high* priority or 1 indicating *low* priority.

Priority expression of mapping must be of type INT

`PRI` expressions may take one of two integer values; either 0 indicating *high* priority or 1 indicating *low* priority.

Process evaluates to STOP

The logic within the configuration description itself evaluates to a STOP and will therefore not execute.

Process mapped at high priority may not contain a PRI PAR

The logic of the `MAP` statement which includes the `PRI` expression conflicts with the internal logic of the program which uses a `PRI PAR`.

Processor type has already been set for NODE *name*

Node *name* already has the `type` attribute set. Check the hardware description.

Processor type has not been set for NODE *name*

The attribute `type` is a mandatory node attribute and should be set in the hardware description for `NODE` *name*.

ROM memory size has not been set for root NODE

The size of ROM attached to the root node must be specified using the `romsize` attribute. (The root node is the node which has the `root` attribute set).

Right hand side of mapping must be a physical NODE

Logical processors are mapped onto a physical processor, with a statement of the following form:

MAP *processor.list* ONTO *node*

where *node* is a physical node declared within the hardware description.

Right hand side of mapping must be of type ARC

Channels are mapped onto an ARC, with a statement of the following form:

MAP *channel.list* ONTO *arc*

where *arc* is a named link declared within the hardware description.

Right hand side of mapping must be of type NODE or ARC

MAP statements are used to map logical processors onto physical processors or channels onto named links called ARCs. The statement takes the form:

MAP *processor.list* ONTO *node*

where *node* is a physical node declared within the hardware description.

OR:

MAP *channel.list* ONTO *arc*

where *arc* is a named link declared within the hardware description.

SET expression must be of type NODE

The SET expression is used to set attributes for physical processors. It takes the form:

SET *device* (*attribute.assignment*)

where *device* is a node name.

Too many channels inputting on ARC *name*

An ARC can carry a maximum of two channels, one in either direction. It is a dedicated link i.e. one which will not be used by virtual channels.

Too many channels outputting on ARC *name*

An ARC can carry a maximum of two channels, one in either direction. It is a dedicated link i.e. one which will not be used by virtual channels.

Unknown attribute name *name*

The attribute *name* used is not supported by the language. Valid attributes are:

`link, linkquota, location.code, location.ws, location.vs, memsize, nodebug, noprofile, order.code, order.ws, order.vs, reserved, romsize, root, routecost, tolerance, type`

Further information can be found in the '*Occam Configuration Language*' appendix in this manual.

Value must be in range 0 to *n* for attribute *name*

Attribute *name* has been incorrectly specified.

WITH expression must be of type ARC

The only configuration statement that a WITH expression may appear in, is a CONNECT statement. This is used to connect an ARC with a link connection. It takes the form:

'CONNECT *edge* TO *edge* [WITH *arcname*]'

5 icollect - code collector

This chapter describes the code collector tool icollect which generates an executable file for a single or multitransputer program from a configuration data file, or for a single transputer program directly from a linked unit. The tool is also used to create files for input to the EPROM programmer tool ieprom, and to create files that can be dynamically loaded by a user program.

5.1 Introduction

icollect generates bootable files for transputer programs, and other executable files in special formats.

Bootable files are transputer executable files that can be directly loaded onto the transputer hardware down a transputer link. The bootable file contains all the information for loading and running the program on a specific network of processors, including data that controls the distribution of code on the network, and self-booting code for each processor. Bootable programs are therefore self-distributing and self-starting when they are sent down a transputer link.

Recommended program development for single and multitransputer programs is to create a configuration data file (i.e. binary file) and to use this as input to the collector. The configuration data file describes the placement of processes and channels on the processor network in a special format which can be read by the collector. They are created from configuration descriptions by the configurer.

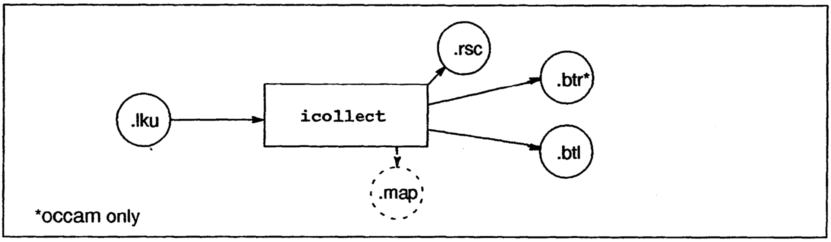
Single transputer programs can by-pass the configuration stage and use a single linked unit as input. The collector then adds bootstrap and system code for a single processor. Unconfigured programs can only run on a single transputer. This method of program development is not recommended and may not be supported in future toolsets.

icollect can be directed to generate output files in a special format for processing by the ieprom tool, and executable code with no bootstrap or system process information, intended for dynamic loading by a supervisory program.

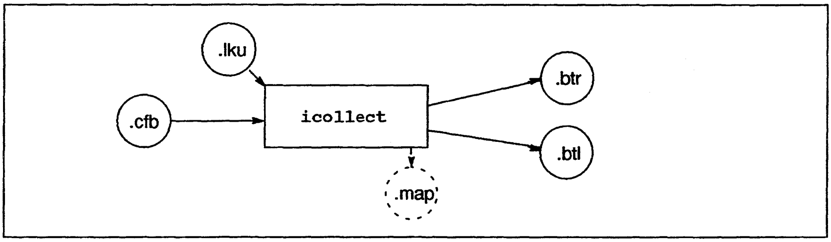
The command line default is to assume input from a configuration binary file. Special format outputs are selected by specifying command line options.

The main inputs and outputs of the collector tool for bootable programs are shown below.

Unconfigured program (using 'T' option):



Configured processor program:



5.2 Running the code collector

The code collector is invoked using the following command line:

► `icollect` *filename* { *options* }

where: *filename* is a configuration data file created by a configurer or a single linked unit created by `ilink`. Only one filename may be given on the command line.

options is a list of the options given in Table 5.1.

> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.
>
> Options may be supplied in an indirect argument file, prefixed by '@'. See section A.1.2 for details.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
BM	Instructs the tool to use a different bootstrapping scheme, which uses the bottom of memory, see section 5.8. This option is only valid for configured programs i.e. when the 'T' option is *not* used.
CM	Instructs the collector to add a bootstrap which will clear memory during the booting and loading of the transputer network. (See section 5.5).
CP	Performs collect time patching i.e. reduces the amount of memory used. (See section 5.10). This option can only be used with the 'T' option (unconfigured mode).
E	Changes the setting of the transputer Halt On Error flag. HALT mode programs are converted so that they not stop when the error flag is set, and non HALT mode programs to stop when the error flag is set. This option can only be used with the 'T' option (unconfigured mode).
I	Displays progress information as the collector runs.
K	Creates a single transputer file with no bootstrap code. If no file is specified the output file is named after the input filename and given the .rsc extension. This option can only be used with the 'T' option (unconfigured mode).
M *memorysize*	Specifies the memory size available (in bytes) on the root processor for single transputer programs. *memorysize* is specified in bytes and may be given in decimal format (optionally followed by 'K' or 'M' to indicate Kilobytes or Megabytes respectively), or it may be specified in hexadecimal using the '#' or '$' prefixes. This option can only be used with the 'T' option (unconfigured mode) and results in a smaller amount of code being produced (see section 5.4).
O *filename*	Specifies the output file. A filename must be supplied and is used as given. (See section 5.2.4).
P *filename*	Specifies a name for the memory map file. A filename must be supplied and is used as given. The file extension .map should be used when the file is to be used as input to imap, see chapter 13.
RA	Creates a file for processing by ieprom into a boot from ROM file to run in RAM. If no output file is specified the filename is taken from the input file and given the .btr extension. This option is only necessary when using the 'T' option (unconfigured mode) to create a ROM code file.
RO	Creates a file for processing by ieprom into a boot from ROM file to run in ROM. If no output file is specified the filename is taken from the input file and given the .btr extension. This option is only necessary when using the 'T' option (unconfigured mode) to create a ROM code file.
RS *romsize*	Specifies the size of ROM on the root processor in bytes. Only valid when used with the 'RA' or 'RO' options. *romsize* is specified in bytes and may be given in decimal format (optionally followed by 'K' or 'M' to indicate Kilobytes or Megabytes respectively), or it may be specified in hexadecimal using the '#' or '$' prefixes. This option is only necessary when using the 'T' option (unconfigured mode) to create a ROM code file.

Option	Description
s *stacksize*	Specifies the extra runtime stack size in words for single transputer programs. (For occam programs this option refers to `stack.buffer`, see section 5.4.2 for details). *stacksize* is specified in words and may be given in decimal format (optionally followed by 'K' or 'M' to indicate Kilowords or Megawords respectively), or it may be specified in hexadecimal using the '#' or '$' prefixes. This option can only be used with the 'T' option.
T	Creates a bootable file for a single transputer. The input file specified on the command line must be a linked unit. This option can not be used for C programs which are linked with the *reduced* runtime library.

Table 5.1 `icollect` command line options

5.2.1 Examples of use

Example A (unconfigured program mode):

```
icc – t450 hello.c
ilink –t450 hello.tco –f cnonconf.lnk
icollect hello.lku –t
```

Example B (configured program mode):

```
icc –st20 hello.c
ilink –st20 hello.tco –f cstartup.lnk
icconf  hello.cfs
icollect hello.cfb
```

```
icc –t805 hello.c
ilink –t805 hello.tco –f cstartup.lnk
icconf  hello.cfs
icollect hello.cfb
```

Note: single transputer programs linked with the *reduced* runtime libraries cannot be linked and collected with the 'T' option, they must be configured.

5.2.2 Default command line

Commonly used command line parameters can can be defined for the tool using the `ICOLLECTARG` environment variable. Parameters specified in this way are automatically added to the command line when the tool is run.

Parameters in `ICOLLECTARG` must be specified using the syntax required by the command line.

5.2.3 Input files

The input file to `icollect` is either a configuration data file generated by a configurer, or a linked unit generated by `ilink`. By default the collector assumes a configuration data file; for single transputer programs the input file may be a linked unit, in this case the 'T' option must be given.

Input files of an incorrect format generate an error message and no output is produced.

5.2.4 Output files

The output produced by the tool depends the type of file input to the collector and the collector options used.

Specifying an output filename

An output filename can optionally be specified using the O option, followed by a filename, which will be used as given. If the O option is not used, the input filename will be used with an extension added indicating the file type, see below.

Default case

The default output file is a binary file that can be loaded directly onto the transputer hardware down a transputer link. This type of file is known as a *boot from link* program. By default the file is given the `.btl` extension.

Boot-from-ROM/RAM

Boot-from-ROM programs output is generated by using the appropriate command line options; RO for boot-from-ROM; RA for boot-from-RAM. By default the file will be given the `.btr` extension.

C programs must be configured, prior to using `icollect`, in order to generate boot-from-ROM or RAM output. occam boot-from-ROM or RAM output can be generated from either configured or unconfigured input.

Dynamically loadable output

Dynamically loadable code is generated by using the K command line option. By default the file is given the `.rsc` extension.

Dynamically loadable code file may only be generated when the input to the collector is a linked unit and the T option is used, i.e. this type of output can only be produced for the unconfigured case.

Memory map files

A memory map file may be generated, in addition to the normal output, by specifying the 'P' option. The format of these files is described in section 5.9.

5.3 Program interface for occam unconfigured programs

For programs which are loaded onto a single transputer, the program interface must conform to the appropriate format, depending on whether or not memory size is specified on the collector command line.

5.3.1 Interface used for 'T' option

occam programs which are collected with the T option, without specifying *memorysize* (using the M option), must use one of the following formats of procedure declaration:

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

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

user.buffer is the free memory buffer.

stack.buffer is a buffer allocated at the base of memory by the collector, whose size is determined by the S option. If the S option is not specified when icollect is invoked this buffer will be of size zero.

5.3.2 Interface used for 'T' and 'M' options

In the case where both the 'T' and 'M' options are used, the program must conform to one of the following procedure declarations:

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

where: The channel protocol can take any valid type.

The other variables are as defined above.

5.4 Memory allocation for unconfigured programs

The memory allocation outlined in this section applies only to single processor programs collected with the 'T' option and without the 'K' option. For configured programs the layout of code and data in memory is determined by the configurer. For programs generated with the 'T' option the layout is determined by the collector. The details of memory use depend on the language used and the options to icollect, this is described below.

Memory which is not reserved by the system for program code and data (known as *free memory*) can be made available to a user application. For C programs this is used for the heap and, optionally, the stack. In the case of a single transputer occam program the free memory passed as an array.

To calculate the actual memory available, the loader program in the bootable file first reads the total memory size from the host environment variable IBOARDSIZE. This communication with the host is performed after the program has been loaded onto the transputer board but before the program is started. The size of the free memory is given by IBOARDSIZE minus the combined program code and data space required.

The process code which reads IBOARDSIZE requires approximately 3.5 Kbytes of memory. This process is executed and terminated before the user program runs, and the segment of free memory that the process uses is then returned to the user program. Therefore when the user program executes it will not know whether the process was present or not.

When the 'M' option is used to specify the memory size, IBOARDSIZE is not read and therefore the total amount of memory required when loading the program will be approximately 3.5 Kbytes less.

A memory map file may be obtained by specifying the 'P' command line option. The content of memory map files is described in section 5.9.

5.4.1 C programs

For C programs the bootstrap loader must allocate memory for static data, stack and heap areas.

When the collector 'S' option is specified the program's stack is placed at the bottom of memory. When the 'S' option is not specified a stack area is allocated by the runtime system, typically at the top of free memory.

Areas for static data and heap are always allocated by the language's runtime system at the bottom of free memory. The heap area grows upwards, towards the top of memory, and the stack grows downwards.

Figure 5.1 shows the memory map layouts for programs with and without the stack requirement specified by the user.

The value of **LoadStart** is described in section 5.9.

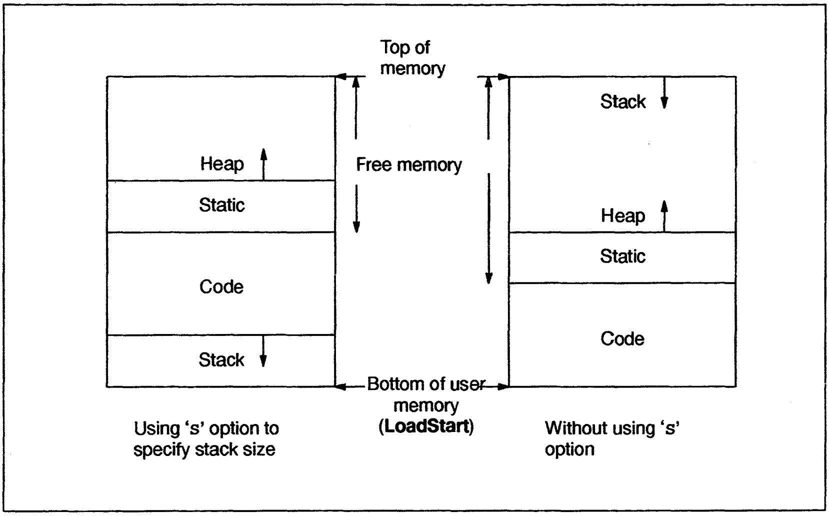


Figure 5.1 Memory maps for C

5.4.2 occam programs

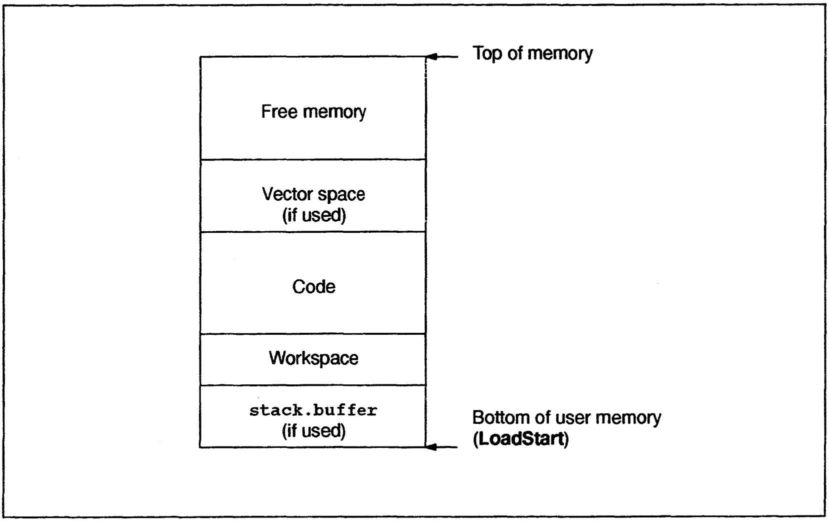


Figure 5.2 Memory map for occam program

An occam program requires space to be allocated for code, workspace and, possibly, vector space. Programs can also be passed one or two arrays as parameters; one (always available) provides access to the free memory. The other is optional but, if used, it is placed at the bottom of the memory map to provide access to the transputer's fast internal RAM. This array is known as the `stack.buffer`. The default bootstrap loader attempts to optimize placement of the program's, and its own, code and workspace. If present, the `stack.buffer` array is placed at the bottom of memory (at **LoadStart**). This is followed in order by the workspace, code, vector space (if used) and free memory. The 'S' option allows space to be reserved in the internal RAM.

Figure 5.2 shows the memory map of the loaded occam code as created by the default bootstrap loader.

5.4.3 Memory initialization errors

While the loader is executing the memory initialization process, described above, warning messages may be obtained which have the following format:

`Warning–SystemA–` *message*

where: *message* can be one of the following:

IBOARDSIZE, unable to read

`IBOARDSIZE` environment variable is not defined correctly.

number, **illegal format number**

The value specified for `IBOARDSIZE` is in the wrong format.

illegal 16 bit memory size, set to zero

The value of `IBOARDSIZE` is greater than 64K when a 16 bit processor is being used. The memory size has therefore been set to zero.

negative memory size, set to zero

A negative value was specified for `IBOARDSIZE`, which has been set to zero.

unable to reset free memory

The loader cannot return the memory it has used to the user.

All the above errors are generated by the system process at runtime.

5.4.4 Small values of IBOARDSIZE

When the 'T' option is used, very small values of `IBOARDSIZE` (including zero) are detected at runtime and prevent the program from being run. `IBOARDSIZE` is read at runtime, not by the collector at build time. Small values of `IBOARDSIZE` cause the collector to generate a warning message but do not prevent the generation of a bootable file.

IBOARDSIZE must be ≥ to the total memory requirements of the user program being executed.

5.5 Clearing memory

The 'CM' collector option, instructs the collector to use a bootstrap that clears memory on each transputer before the application code is executed. The bootstrap will clear the memory for all processors in the network.

In order to clear the memory on a processor, it is necessary for the bootstrapping sequence to know the size of the memory. There are four cases to consider:

1 **A configured program.**

Here the memory size is known at configuration time, and is specified by the user in the configuration source file. The bootstrapping sequence produced by the collector will clear the amount of memory specified (in the configuration source file) before booting the application.

2 **A collected program with fixed memory size.**

The collector may be used, with the 'T' option, to produce a bootable file from a single linked unit. The amount of memory on the processor may be specified with the 'M' option. In this case the bootstrapping sequence will clear the amount of memory specified (with the 'M' option) before booting the application.

3 **A collected program with variable memory size, booted from link.**

If the collector is run with the 'T' option, but without the 'M' option, the memory size is known only at runtime. The memory size is found out at runtime using the environment variable IBOARDSIZE. In this case the bootstrapping sequence will clear memory up to the minimum required to boot the program. After booting, the value of IBOARDSIZE will be read and the remaining memory will be cleared.

4 **A collected program with variable memory size, booted from ROM.**

If the collector is run with the 'T' option, but without the 'M' option, and the program is booted from ROM, then the memory size is not known at all. In this case the bootstrapping sequence will clear enough memory for the minimal requirements of the application. It is then the user program's responsibility to clear any additional memory required.

5.6 Non–bootable files created with the K option

Files created with the 'K' option are non-bootable files which can be dynamically loaded or manipulated by a program at runtime. Non-bootable files cannot be loaded and run on transputer hardware in the normal way.

5.6.1 File format

Non-bootable files consist of program code preceded by a specific sequence of data words which provide runtime information. The sequence of data words and code blocks is summarized in table 5.2. Descriptions of the more important data items are given after the table.

Data	Number of bytes occupied	Unit
Interface descriptor size	Four	bytes
Interface descriptor	Set by above	–
Compiler id size	Four	bytes
Compiler id	Set by above	–
Target processor type	Four	–
Version number	Four	–
Program scalar workspace requirement	Four	words
Program vector workspace requirement	Four	words
Static size	Four	words
Program entry point offset	Four	bytes
Program code size	Four	bytes
Program code block	Set by above	–

Table 5.2 Sequence of code segments in non-bootable files

Target A value indicating the processor type or transputer class for which the program was compiled. Set by compiler options or by default. Possible values and their meaning are:

Value	Applies to:
2	T212, T222, T225, M212
4	T414
8	T800, T801, T805
9	T425, T400
10	TA
11	TB
1024	T450, ST20

Version The format version number of the file. This can be either 10 or 11 in TCOFF files. For C programs this value is 11, which indicates that the 'Static size' parameter (below) is present. For occam programs the value is 10, indicating no static data; the parameter list will also not be present.

Scalar workspace Specifies the size of the workspace required for the linked program's runtime stack.

Vector workspace Specifies the size of the workspace required for the linked program's vector (array) data.

Static size Specifies the size of the static area (only present if the file format version number is 11).

Entry point offset Indicates the offset in bytes of the program entry point from the base of the code block.

Code size	Indicates the size of the program code in bytes.
Code	The program code.

5.7 Boot-from-ROM output files

Boot–from–ROM output files are either generated by using the collector options 'RA' or 'RO' for unconfigured programs or by configuring a program to boot from ROM, prior to collecting. (The configurer also has 'RA' and 'RO' command line options).

The boot-from-ROM files contain code that can be loaded into EPROM using the `ieprom` tool. The code may be run on the root transputer of a network; processors on the network connected to the root transputer are booted from the root transputer's links.

'RA' generates code which is executed from RAM.The code is copied from ROM into RAM at runtime. 'RO' generates code which is directly executed from ROM.

RAM executable code can be used for applications which are to be executed from fast RAM, and for code which may be user-modified. ROM executable code may require no external RAM for programs which use small amounts of data and can be used to create a truly embedded system.

5.8 Alternative bootstrap schemes

When building for a configured network, the collector uses a bootstrapping scheme which makes use of the top two hundred bytes of memory. This memory is required to load the last few bytes of application code prior to its execution. The memory region becomes available to the user once their application is running.

This scheme does not remove memory from the user's environment on a permanent basis and it facilitates the absolute placement of code and data by the user. See the '*User Guide*' for details.

The user can tell the collector to use a different booting scheme by using the option 'BM'. In this case the booting scheme permanently removes a section of memory from the user's environment and moves the value of **LoadStart** accordingly. This section of memory is never made available to the user. This booting scheme does not support the absolute placement of code and data by the user.

The booting scheme invoked by the 'BM' option, is used by default for unconfigured programs i.e. those collected using the 'T' option.

5.9 The memory map file

A memory map file may be obtained by specifying the 'P' command line option, followed by a filename. Such files contain the memory layout for each processor in the network.

The file layout takes the form of a list of code and data to be placed on respective processors. The right hand side of the file gives the start and end address followed by the size of each block.

The memory map file contains the following information:

- icollect version data
- For each processor the following details are given:
  - ○ Processor type
  - ○ Error mode (HALT or STOP)
  - ○ **LoadStart** (lowest user memory address)
  - ○ For each process on this processor the following is listed:
    - □ Code, name of file, offset from start (decimal), start address and end address (hex), size (decimal), entry address (if any, in Hex)
    - □ Workspace, start and end address (hex), size (decimal)
    - □ Any other data requirements
- Boot path for the network - only present if program is configured
- Connectivity of the network - only present if program is configured

The absolute addresses are calculated using **LoadStart**, which is the base of user memory. This varies for different processor types i.e. the value of **LoadStart** for a T4 processor is different to that for a T8.

If the 'BM' option is used the memory from **MemStart** to **LoadStart** is used by the low level bootstraps and their workspace.

When the 'BM' option is not used the value of **LoadStart** is determined by the configuration, see the reference chapter for the configurer, for further details.

The addresses allocated to various data items reflect the command line options specified to the collector. Details of the memory map files for the following types of files are given below:

- Unconfigured (single processor), boot from link programs targetted at a specific processor type.
- Unconfigured (single processor), boot from link programs targetted at a processor class.
- Configured, boot from link programs.
- Boot from ROM (single and configured)

The examples below demonstrate the map file format; they may change in detail.

5.9.1 Configured program boot from link

```
icollect : INMOS toolset collector, Sun Version 3.0.29


Memory map for 'Single' processor 0 ST20
Load Start is 80000170,  HALT ON ERROR,  Minimum memory size is 68488

  HIGH priority INITSYSTEM process 'Init.system.simple'
    Code from 'sysproc.lib', file offset 19736
                                        #800001B8  #80000240       136
      Entry address                     #800001BB
    Invocation stack                    #80000194  #800001B8        36
    Workspace                           #80000170  #80000194        36

  HIGH priority OVERLAID SYSTEM process 'System.process.b'
    Code from 'sysproc.lib', file offset 143550
                                        #80000274  #800002E4       112
      Entry address                     #80000277
    Invocation stack                    #80000260  #80000274        20
    Workspace                           #80000240  #80000260        32

  LOW priority USER process 'Simple'
    Code from 'hello.lku', file offset 2
                                        #80001198  #80003E38     11424
      Entry address                     #800011B4
    Invocation stack                    #80001184  #80001198        20
    Workspace                           #80000170  #80001184      4116
    Static                              #80003E38  #80004198       864
    Heap                                #80004198  #80010998     51200

  Parameter data                        #80010998  #80010AC4       300


Boot path for network

  Boot processor 0 down link 0 from HOST


Connectivity for network

  Connect HOST to processor 0 link 0
```

Figure 5.3 Memory map file for a configured ST20 processor program

The example shown in Figure 5.3 was produced by the following command line:

```
icollect hello.cfb -p hello.map
```

where: `hello.cfb` is the configuration binary file produced by the configurer for the single processor 'Hello World' example program.

The Memory map for the configured program is similar to that produced for unconfigured transputer programs (see section 5.9.2) except that it has two additional configuration sections at the end of the file. The *Boot path* for the network lists processors in the order in which they are to be booted. The *Connectivity for network* lists the link connections between the processors.

5.9.2 Unconfigured (single processor), boot from link

Program targetted at transputer type

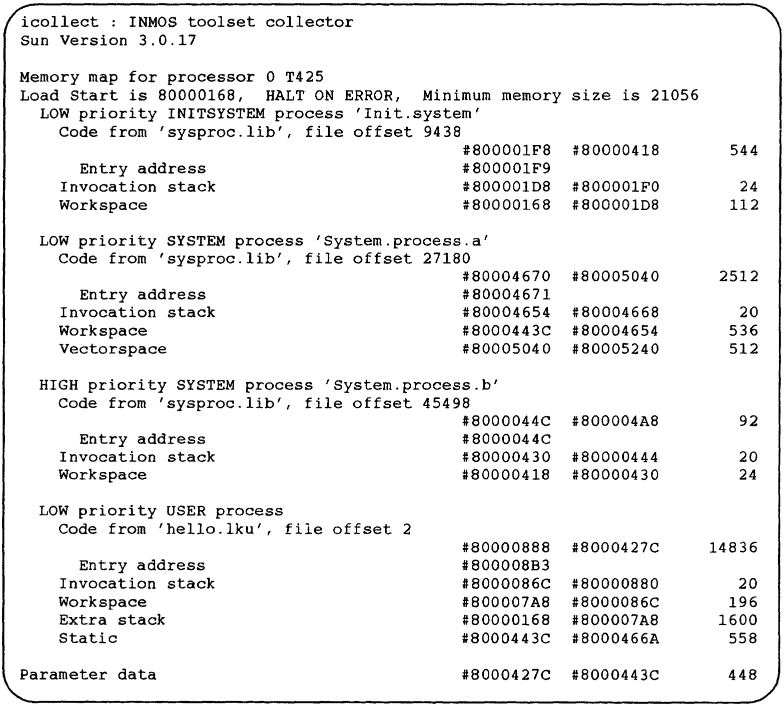
The first memory map described in this section is for a program which is to be booted for a specific processor type.

The example shown in Figure 5.4 was produced by the following command line:

```
icollect -t hello.lku –s 400 -p hello.map
```

where: `hello.lku` was produced by compiling and linking the example program `hello.c` for a T425 in the default halt-on-error mode. The compiled object file was linked with the C linker indirect file `cnonconf.lnk` because the example is for an unconfigured program.

`hello.map` lists code and data segments to be placed on each processor (one in this case). For each process the workspace and vector space requirements are given together with the entry point of the process. Note that the first three processes listed are non-user processes; this will always be the case for this type of program.



```
icollect : INMOS toolset collector
Sun Version 3.0.17

Memory map for processor 0 T425
Load Start is 80000168,  HALT ON ERROR,  Minimum memory size is 21056
  LOW priority INITSYSTEM process 'Init.system'
    Code from 'sysproc.lib', file offset 9438
                                            #800001F8  #80000418      544
      Entry address                         #800001F9
    Invocation stack                        #800001D8  #800001F0       24
    Workspace                               #80000168  #800001D8      112

  LOW priority SYSTEM process 'System.process.a'
    Code from 'sysproc.lib', file offset 27180
                                            #80004670  #80005040     2512
      Entry address                         #80004671
    Invocation stack                        #80004654  #80004668       20
    Workspace                               #8000443C  #80004654      536
    Vectorspace                             #80005040  #80005240      512

  HIGH priority SYSTEM process 'System.process.b'
    Code from 'sysproc.lib', file offset 45498
                                            #8000044C  #800004A8       92
      Entry address                         #8000044C
    Invocation stack                        #80000430  #80000444       20
    Workspace                               #80000418  #80000430       24

  LOW priority USER process
    Code from 'hello.lku', file offset 2
                                            #80000888  #8000427C    14836
      Entry address                         #800008B3
    Invocation stack                        #8000086C  #80000880       20
    Workspace                               #800007A8  #8000086C      196
    Extra stack                             #80000168  #800007A8     1600
    Static                                  #8000443C  #8000466A      558

Parameter data                              #8000427C  #8000443C      448
```

Figure 5.4 Memory map file for a single T425 processor program

Program targetted at transputer class

The second memory map described in this section is for a program which is to be booted for processor classes TA or TB.

The example shown in Figure 5.5 was produced by the following command line:

```
icollect -t hello.lku -p hello.map
```

where: **hello.lku** was produced by compiling and linking the example program **hello.c** for class TA in the default halt-on-error mode. The compiled object file was linked with the C linker indirect file **cnonconf.lnk** because the example is for an unconfigured program.

```
icollect : INMOS toolset collector
Sun Version 3.0.17

Memory map for processor 0 TA
Load Start is UNKNOWN,  HALT ON ERROR,  Minimum memory size is 20180
  LOW priority INITSYSTEM process 'Init.system'
    Code from 'sysproc.lib', file offset 10420
                                                #3D48      #3F68      544
      Entry address                             #3D49
    Invocation stack                            #3D28      #3D40       24
    Workspace                                   #3CB8      #3D28      112

  LOW priority SYSTEM process 'System.process.a'
    Code from 'sysproc.lib', file offset 30561
                                                #419C      #4B6C     2512
      Entry address                             #419D
    Invocation stack                            #4180      #4194       20
    Workspace                                   #3F68      #4180      536
    Vectorspace                                 #4B6C      #4D6C      512

  HIGH priority SYSTEM process 'System.process.b'
    Code from 'sysproc.lib', file offset 45888
                                                  #34        #90       92
      Entry address                               #34
    Invocation stack                              #18        #2C       20
    Workspace                                      #0        #18       24

  LOW priority USER process
    Code from 'hello.lku', file offset 2
                                                  #E0      #3AF8    14872
      Entry address                              #10B
    Invocation stack                              #C4        #D8       20
    Workspace                                      #0        #C4      196
    Static                                      #3F68      #4196      558

Parameter data                                  #3AF8      #3CB8      448
```

Figure 5.5   Memory map file for a single TA processor program

The memory layout of user's code and data is the same as for the previous example, except that no space is allocated for the extra stack (because extra stack was not requested on the command line). **LoadStart**, from which the start and end addresses are calculated, can only be calculated at runtime. This is because the value of **MemStart** cannot be determined at collect time. The numbers given, in place of absolute addresses are offsets from **LoadStart.**

5.9.3 Boot from ROM programs

There are four cases for this type of program:

- Unconfigured (single processor), boot from ROM, run in RAM
- Unconfigured (single processor), boot from ROM, run in ROM
- Configured program, boot from ROM, run in RAM
- Configured program, boot from ROM, run in ROM

The memory maps for each of these are summarized below.

Unconfigured (single processor), boot from ROM, run in RAM

The memory map for this case will have the same layout as the single processor boot from link programs.

Unconfigured (single processor), boot from ROM, run in ROM

It is not known at collect time where in memory the ROM is to be placed. Therefore, the start and end addresses of the code segments are given as offsets from the start of ROM, and are annotated as such. Items such as workspace will have absolute addresses allocated, if the program is targetted at a specific processor type.

Note: for C programs the runtime startup system would require modification first, in order to provide the system with details of heap and stack etc.

Configured program, boot from ROM, run in RAM

The layout of the memory map for this case will be the same as that for the boot from link configured program. This is because everything (code and data) is copied into RAM.

Configured program, boot from ROM, run in ROM

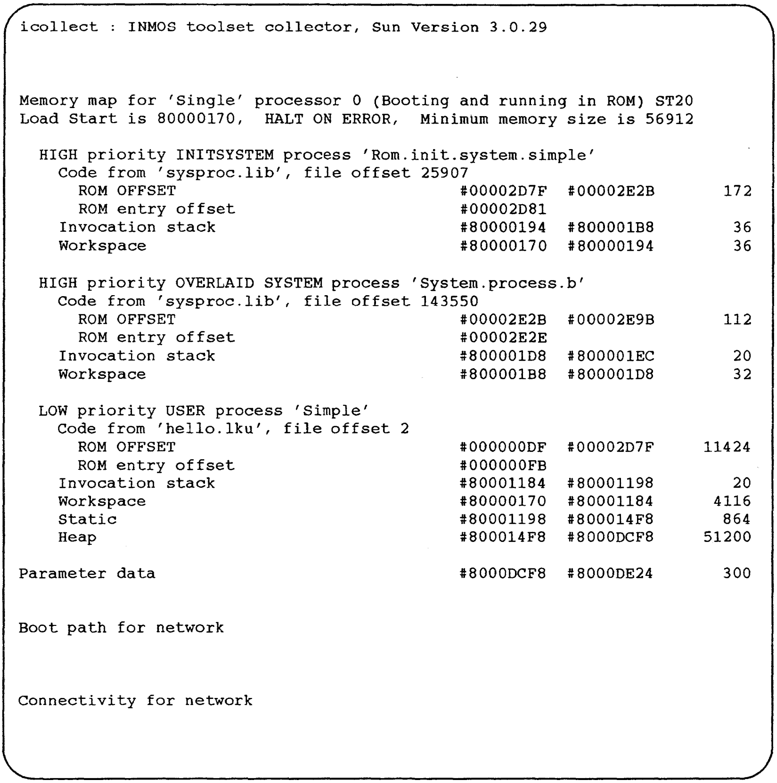
For this case the root processor will be shown in the same format as the single processor case, run in ROM; some memory locations being expressed as offsets from the beginning of ROM.

The other processors in the network will appear as in the boot from link case.

The example shown in Figure 5.6 was produced by the following command line:

```
icollect hello.cfb -p hello.map
```

where: **hello.cfb** is the configuration binary file produced by the configurer, for the single processor 'Hello World' example program. The configurer 'RO', 'RS' and 'P' options were used to create a boot from ROM input file for the collector.



```
icollect : INMOS toolset collector, Sun Version 3.0.29



Memory map for 'Single' processor 0 (Booting and running in ROM) ST20
Load Start is 80000170,  HALT ON ERROR,  Minimum memory size is 56912

  HIGH priority INITSYSTEM process 'Rom.init.system.simple'
    Code from 'sysproc.lib', file offset 25907
      ROM OFFSET                              #00002D7F  #00002E2B       172
      ROM entry offset                        #00002D81
    Invocation stack                          #80000194  #800001B8        36
    Workspace                                 #80000170  #80000194        36

  HIGH priority OVERLAID SYSTEM process 'System.process.b'
    Code from 'sysproc.lib', file offset 143550
      ROM OFFSET                              #00002E2B  #00002E9B       112
      ROM entry offset                        #00002E2E
    Invocation stack                          #800001D8  #800001EC        20
    Workspace                                 #800001B8  #800001D8        32

  LOW priority USER process 'Simple'
    Code from 'hello.lku', file offset 2
      ROM OFFSET                              #000000DF  #00002D7F     11424
      ROM entry offset                        #000000FB
    Invocation stack                          #80001184  #80001198        20
    Workspace                                 #80000170  #80001184      4116
    Static                                    #80001198  #800014F8       864
    Heap                                      #800014F8  #8000DCF8     51200

Parameter data                                #8000DCF8  #8000DE24       300


Boot path for network



Connectivity for network
```

Figure 5.6 Memory map for program configured to boot from and run in ROM

5.10 Reducing the amount of memory used – 'Y' option

The 'Y' collector option reduces the amount of memory used.

For programs compiled and linked for a specific transputer type, this option will cause `icollect` to produce a program that uses less memory. However, programs compiled and linked for transputer classes 'TA' or 'TB' will not build when this option is used. This option is only valid for programs collected with the T option.

5.11 Error messages

This section lists error messages generated by icollect. The messages are listed in alphabetical order under the appropriate severity classification. In all cases the introductory string (severity, and filename if appropriate) is omitted.

icollect generates errors of severities *Warning* and *Serious*. Serious errors cause the tool to terminate without producing any output.

5.11.1 Warnings

The following messages are prefixed with 'Warning-'. They are only generated when the 'T' option is used (single processor mode).

Flip error mode ignored with user bootstrap

> The 'E' option is ignored when a user-defined bootstrap is specified since the collector will only accept a single linked unit as a bootstrap.

Strange board size for sixteen bit processor: Setting to zero

> The memory size specified exceeds the addressing capacity of a 16 bit processor (64 Kbytes). The collector uses a memory size of zero for the rest of the build.

5.11.2 Serious errors

The following errors are prefixed with 'Serious-'.

Address space for target processor exhausted

> The address space required by the program is greater than 64Kbytes, the maximum addressable space on a 16-bit processor.

Cannot have both rom types

> 'RA' and 'RO' options are mutually exclusive and cannot both be specified on the same command line.

Cannot have configured and memory size

> The memory size option is incompatible with building a bootable program for a configuration binary file.

Cannot have configured and non bootable file

> The collector cannot generate both a network loadable file and a non-bootable file simultaneously for the same program.

Cannot have rom and non bootable file

> The collector cannot generate both a ROM-loadable file and a non-bootable file simultaneously for the same program.

Cannot open file *filename*

Host file system error. The file specified cannot be opened.

Cannot patch parameter data for processor class

The 'Y' option has been specified with a linked unit for a processor class. The collector cannot initialize some of the data without a linked unit for a specific processor type.

Cannot use absolute placement and bottom of memory loader

The user has specified BM to the collector but is using absolute code and data placement at configuration. This combination is not legal.

Command line parsing error at *string*

Unrecognized command line option.

Dynamic memory allocation failure

Memory allocation error. The collector cannot allocate the required amount of memory for its internal data structures.

Expected end tag found not present in .cfb file

A specific end tag is missing in the configuration binary file. Either the file is corrupted or the versions of `icollect` and configurer used are incompatible.

Illegal tag found in .cfb file

Incorrect format configuration binary file, recognized as an illegal tag. Either the file is corrupted or the versions of `icollect` and configurer used are incompatible.

Illegal language type found in input file

Source language used to create the file is not supported by the collector. Less likely, but possible, is that the file was created using an incompatible (possibly earlier) version of a tool.

Illegal process type

Unrecognized process type. Either the file has been corrupted or the versions of `icollect` and configurer used are incompatible.

Illegal processor type

Unrecognized processor type. Either the file has been corrupted or `icollect` and the configurer are incompatible.

Illegal tag found in input file : *filename*

Incorrect format input file. The most likely reason for this error is that an incorrect file has been specified. Other less likely but possible explanations are that the file was created using an earlier or incompatible version of one of the tools, or that the file has become corrupted.

Input file already specified

More than one input file specified on the command line.

Input file has not been linked *filename*

The collector accepts only linked files, either directly when using single processor operation, or indirectly via entries in the configuration binary file. This message can be generated if the file was created using a previous version of a tool, or if the file is corrupt.

Input file is of incorrect type: *filename*

If the '`T`' option is specified (single processor program) the input file must be a single linked unit (`.lku` type). If the '`T`' option has not been specified the input file must be a configuration binary file (`.cfb` type).

Input filename too long

The maximum length allowed for the input filename is 256 characters.

Linked unit file in cfb and linked unit in input file found do not match: *filename*

The linked file specified in the configuration binary and the one found the collector do not match.

Linked unit module not found in: *filename*

The required library module is missing or has been corrupted. This message is generated when an incorrect version of the library is installed.

Memory requirement for build is greater than specified, an extra *<n>* **bytes required at least**

The amount of memory specified on a processor is not enough for the program to execute. An extra *<n>* bytes are required at least.

Memory size already specified

Memory size must be specified once only.

Memory size string invalid

Memory size must be given in decimal or hex. Hex numbers must be introduced by '`#`' or '`$`'.

Memory size string too long

Specified memory size is too large.

More than one parameter statements

The collector expects only one *parameter* statement per processor. Either the file has been corrupted or the versions of `icollect` and configurer used are incompatible.

No input file specified

One, and only one, input file must be specified on the command line.

No parameter descriptor present in input file: *filename*

The formal parameter descriptor in the input file is not present. This usually means that the process has not been linked with a main entry routine. This message will only appear if the collector is invoked with the `T` option (unconfigured mode).

Output file already specified

More than one output file was specified. Specify only one.

Output filename too long

The maximum length allowed for the output filename is 256 characters.

Parameter descriptor error in input file : *filename*

The formal parameter descriptor in the input file is not of the correct form, indicating that the process interface is not one recognized by the collector. This message will only appear if the collector is invoked with the `T` option (unconfigured mode). See section 5.3.

Print map file already specified

More than one print map file was specified. Specify one only.

Program configured for boot from ROM command line is boot from link

The specified configuration binary file was created for either ROM or RAM, and neither has been specified to `icollect`.

Program configured for running in RA mode command line is RO mode

Wrong mode specified, or incorrect option given to the configurer when the specified configuration binary file was created. RA and RO modes are mutually exclusive.

Program configured for running in RO mode command line is RA mode

Wrong mode specified, or incorrect option given to the configurer when the specified configuration binary file was created. RA and RO modes are mutually exclusive.

Require at least *<ny>* **bytes at the top of memory for bootstrapping on processor** *<n>*

The bootstrapping sequence requires an extra *<ny>* bytes at the top of memory. Once the bootstrapping has finished this memory is available to the user.

Rom size already specified

ROM size must be specified once only.

Rom size in input file and command line do not match

The ROM size specified on the command line does not match that specified to the configurer when the input file was created.

Rom size not specified

A ROM size must be specified because the input file is to be loaded into ROM.

Rom size string invalid

ROM size must be given in decimal.

Rom size string too long

ROM size specified was too large.

Stack size already specified

Stack size must be specified once only.

Stack size string invalid

Stack size must be specified in decimal format.

Stack size string too long

Specified stack size was too large.

Strange function or attribute for linked unit in : *filename*

The collector has found an unfamiliar value in the input file. Either an old version of a tool was used in the creation of the input file, or the input file has been corrupted.

System error

Host system error has occurred, probably when accessing a file. This message may be generated when a file is read and its contents seem to have changed or the file does not exist.

Unexpected end of file : *filename*

One of the files specified in the configuration binary has ended prematurely. *filename* identifies the offending file. If the message 'Suspect corrupted file' is substituted for *filename* then the file is corrupted.

5.11.3 Fatal errors

Internal error *<message text>*

An internal error has occurred this should be reported to your local SGS-THOMSON distributor or field applications engineer.

6 iemit - memory interface configurer

This chapter describes the memory configuration tool iemit. This tool can be used interactively to explore the effects of changes in the external memory interface parameters of certain 32 bit transputers. The tool can also be used in batch mode to create ASCII or PostScript files. The tool produces a memory configuration file which may be included as an input file to ieprom and blown into EPROM along with a ROM-bootable application file.

The chapter describes how to use iemit and outlines its capabilities. Example displays are provided, followed by a list of error messages which the tool may generate. The format of the memory configuration file is described and an example is given. **Note**: memory configuration files are simple text files which may be created manually using a standard text editor or generated by using iemit.

6.1 Introduction

The IMS T400, T414, T425, T800 and T805 transputers have a configurable external memory interface (EMI) which allows a variety of types of memory device to be connected using few extra components.

For these transputers, the interface configuration may be selected by one of two mechanisms. The user may select one of the 17 standard memory configurations (13 for the T414) or a customized memory configuration may be loaded from a ROM or PAL on reset.

Both methods of memory configuration are available when booting from ROM or from link. If the transputer is being booted from ROM, a customized memory configuration may be added to the ROM or a standard configuration may be used. If the transputer is booted from link a standard configuration may be used at no extra cost, or a dedicated ROM or PAL may be added for a customized configuration.

In order to generate a customized configuration the user may create a memory configuration file, describing the memory configuration and have this blown into an EPROM. The configuration chosen is made known to the transputer by simple board level connections which are detected by the transputer on reset. If a standard configuration is required the **MemConfig** pin is connected to the appropriate address pin. For example, standard configuration 7 is selected via address pin **MemAD7**. If a customized configuration is required the **MemConfig** pin is connected though an invertor to the appropriate data line, usually this is **MemnotWrD0**. **Note**: when iemit is used to generate the memory configuration, the **MemnotWrD0** pin must be used. For further details see the relevant device datasheet.

The external memory interface configuration tool iemit produces timing diagrams for potential configurations of the memory interface and warns of possible errors in the design. It indicates whether one of the preset configurations that are available would be suitable, or whether it would be necessary to use an externally programmed configuration.

Note: That it is assumed that readers creating memory configuration files are familiar with the memory interface of the processor that they are using. The stages in designing a memory interface, including examples, are described in chapter 2 of *The transputer applications notebook - Systems and performance.* Further information may also be found in *The transputer databook.*

6.2 Running iemit

The iemit tool can be invoked by the following command line:

```
▶    iemit options
```

where: *options* is a list of options given in Table 6.1.

Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

Options may be supplied in an indirect argument file, prefixed by '@'. See section A.1.2 for details.

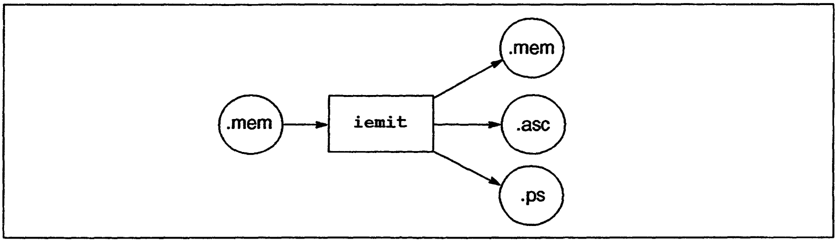
If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
A	Produce ASCII output file.
E	Invoke interactive mode.
F *filename*	Specify input memory configuration file.
I	Select verbose mode. In this mode the user will receive status information about what the tool is doing during operation for example, reading or writing to a file.
O *filename*	Specify output filename.
P	Produce PostScript output file.

Table 6.1 iemit command line options

Note: that if option 'E' is selected i.e. interactive mode, then *no* other options may be specified on the command line.

The operation of iemit in terms of standard file extensions is shown below:



Examples of use

`iemit` may be invoked in interactive mode by using one of the following command:

```
iemit -e
```

Output files in ASCII or PostScript may be specified by command options from within interactive mode; alternatively `iemit` may be invoked in batch mode, to create an output file in one of these formats.

When the tool is invoked in batch mode to produce an output file in either ASCII or PostScript format, then an input file must be supplied using the 'F' option. It is also mandatory to specify either the 'A' or 'P' option. If the 'O' parameter is not supplied then an output filename will be constructed, from the input filename, with an extension of '.ps' for a PostScript output, or '.asc' for an ASCII output.

Example:

The following command causes `iemit` to produce an output file in PostScript format. The tool is invoked in verbose mode.

```
iemit –i –p –f memconfig.mem –o waveform.ps
```

Note: `iemit` will make use of the `ITERM` host environment variable, if it is available, otherwise it will use defaults.

6.3 Output files

Two different types of output may be produced by `iemit`, these are listed below:

- A memory configuration file suitable for including as an input file to the `ieprom` tool.
- An output file in either ASCII or Postscript format, suitable for inclusion in documentation.

The tool may be used interactively to produce a memory configuration file in text format. This file may then be used as an input file to the `ieprom` tool, thus enabling the memory configuration to be stored on ROM. `iemit` is capable of saving and reloading configurations to allow for design over an extended period and for comparison of different configurations. The memory configuration file is described and an example is given in section 6.6.

Additionally `iemit` may be used to produce an output file which is either a plain ASCII file containing timing data or a file in PostScript format containing waveform diagrams. These formats were chosen so that the results of the program could be easily included in reports or other documentation.

6.4 Interactive operation

When `iemit` is invoked in interactive mode the program will start up with the default standard configuration 31.

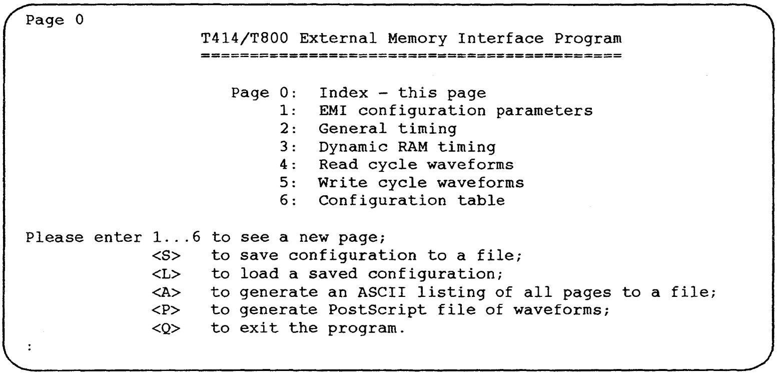
The tool's user interface is presented as a number of display pages showing timing data. The displays may be updated by changing the timing parameters, which are accessed from page 1. All inputs are executed immediately so that the user can see the effect on any of the displays. As each page is shown, the user has the option of selecting another page for display by keying in its number. The current configuration may be saved at any time to a specified output file.

The information displayed and options available on each page are described below.

6.4.1 Page 0

This page acts as an index to the others. It shows the title of each page and allows one of them to be selected. An option is provided to enable an input file to initialize the memory configuration. Other options enable the user to selectively generate output files. Options are listed in table 6.2 and an example of the display page is given in figure 6.1.

The user enters an option code followed by the RETURN key. If a file option is specified the user will be prompted for a filename. **Note**: file extensions should be specified, there are no defaults.



```
Page 0
                          T414/T800 External Memory Interface Program
                          ===========================================

                             Page 0:  Index - this page
                                  1:  EMI configuration parameters
                                  2:  General timing
                                  3:  Dynamic RAM timing
                                  4:  Read cycle waveforms
                                  5:  Write cycle waveforms
                                  6:  Configuration table

Please enter 1...6 to see a new page;
             <S>   to save configuration to a file;
             <L>   to load a saved configuration;
             <A>   to generate an ASCII listing of all pages to a file;
             <P>   to generate PostScript file of waveforms;
             <Q>   to exit the program.
:
```

Figure 6.1 Example `iemit` display page 0

Option	Description
1 to 6	Selects the page to be displayed.
Q	Quit - selection of this option exits the program.
L	Load previously saved configuration. A filename is prompted for, and the configuration saved in that file is read in and the display data is updated. The program expects a memory configuration file. If loading does not succeed because the file has a bad format, the current configuration is reset to standard configuration 31. If loading fails because the file could not be found or could not be opened for reading, the load is abandoned without losing the current configuration.
S	Save configuration to a file. The program prompts for the name of a file to which the data will be written, by convention the extension `.mem` should be used. Output is a memory configuration file. An error is reported if the data could not be saved. The saved file is given comments in its header indicating that it was created by the `iemit` program.
A	Output pages in ASCII format to a file. The program prompts for the name of a file to which the data will be written. Output is in plain ASCII format with a form feed (FF) character after each page. It includes full timing information and a representation of the timing diagrams for read and write cycles. An error is reported if the output could not be written.
P	Generate PostScript file. The program prompts for a filename. The program writes to the file a program in the PostScript page description language to draw the timing diagrams for the chosen memory interface configuration. The waveforms shown are the same as those which can be seen by selecting pages 4 and 5. The file produced fully conforms to the PostScript structuring conventions, allowing it to be processed by other programs. The diagram is designed to fit lengthways on an A4 page, and is suitable for inclusion in technical notes and reports. The file can be sent directly to an Apple LaserWriter or other PostScript output device.

Table 6.2 `iemit` page 0 options

6.4.2 Page 1

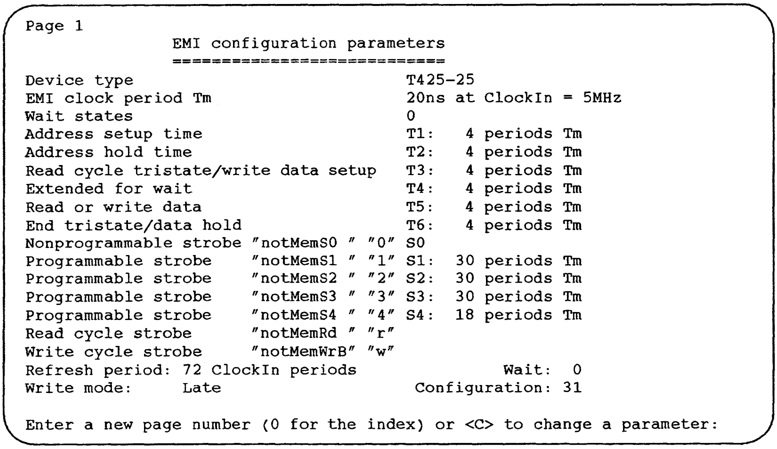
This page shows the input parameters to `iemit`. It is from these parameters that the tool computes the timing information and the waveforms. Only one parameter may be changed at a time and the display data is immediately updated. An example of the display page is given in figure 6.2.

When the page is displayed, the user has the option to select a new page by entering its number, or entering C to change one of the parameters. In the latter case, a list of parameter identifiers is displayed (see table 6.3) and the user is a prompted to select one. The user may then specify a new value, or by pressing the RETURN key, leave the current selection unchanged. The parameters used for modifying the timing data are described in table 6.4.

Note: there are two parameters displayed on page 1 which are calculated by `iemit` and cannot be directly updated by the user; they are the EMI clock period Tm and the Wait states (see Table 6.4).

Parameter identifier	Parameter
0 to 6	Page to be displayed
D	Device type
T1	Address setup time before address valid strobe
T2	Address hold time after address valid strobe
T3	Read cycle tristate or write data setup
T4	Extendible data setup time
T5	Read or write data
T6	End tristate or data hold
S0	Non-programmable strobe **"notMemS0"**
S1	Programmable strobe **"notMemS1"**
S2	Programmable strobe **"notMemS2"**
S3	Programmable strobe **"notMemS3"**
S4	Programmable strobe **"notMemS4"**
RS	Read cycle strobe name
WS	Write cycle strobe name
R	Refresh period
WM	Write mode
W	Memwait input connection
C	Standard configuration

Table 6.3 `iemit` page 1 parameter identifiers



```
Page 1
               EMI configuration parameters
               ============================
Device type                            T425-25
EMI clock period Tm                    20ns at ClockIn = 5MHz
Wait states                            0
Address setup time                     T1:   4 periods Tm
Address hold time                      T2:   4 periods Tm
Read cycle tristate/write data setup   T3:   4 periods Tm
Extended for wait                      T4:   4 periods Tm
Read or write data                     T5:   4 periods Tm
End tristate/data hold                 T6:   4 periods Tm
Nonprogrammable strobe "notMemS0 " "0" S0
Programmable strobe    "notMemS1 " "1" S1:  30 periods Tm
Programmable strobe    "notMemS2 " "2" S2:  30 periods Tm
Programmable strobe    "notMemS3 " "3" S3:  30 periods Tm
Programmable strobe    "notMemS4 " "4" S4:  18 periods Tm
Read cycle strobe      "notMemRd " "r"
Write cycle strobe     "notMemWrB" "w"
Refresh period: 72 ClockIn periods              Wait:  0
Write mode:     Late                    Configuration: 31

Enter a new page number (0 for the index) or <C> to change a parameter:
```

Figure 6.2 Example `iemit` display page 1

Parameter	Description
Device type	This parameter enables the program to deduce the time taken for a half cycle of the signal **ProcClockOut**: this is **Tm**, the basic unit of time of the memory interface. A menu of the available devices is displayed and the user is invited to select one: T400–20 T414–15 T414–17 T414–20 T425–17 T425–20 T425–25 T425–30 T800–17 T800–20 T800–22 T800–25 T800–30 T800–35 T805–25 T805–30
Tstates Tl–T6	The length of each Tstate T1 to T6, is entered as a number of Tm periods between 1 and 4. (2 Tm periods = 1 clock cycle).
Programmable Strobes S0–S4	The programmed durations of the strobes **notMemS0** to **notMemS4**. The strobes each have two names which can be altered. One which can be up to 9 characters in length, and one consisting of just one character. There should be no embedded spaces in the long names. The short names are used in the timing information on pages 2 and 3, while the long names are used to label the waveforms on pages4 and 5, and in the PostScript output. The signal names are initialized to sensible defaults. **Note**: that S0 is a fixed strobe, so its duration cannot be changed. The duration of a strobe can be 0 to 31 **Tm** periods. If the value for S1 is set to zero, then **notMemS1** stays high throughout the cycle; if the value for S2, S3 or S4 is set to zero, then the strobe is low for the duration of the cycle.
Read strobe name	The names for the read strobe **notMemRd** can be altered.
Write strobe name	The names for the write strobe **notMemWrB** can be altered. Note that because the four byte write strobes have the same timing, only one is considered.
Refresh period	The refresh period is given as a number of **ClockIn** periods (18, 36, 54, or 72) or as **Refresh Off** if zero is selected.
Write mode	The write mode can be set to **Early** or **Late** to suit the type of memory being used.
Wait connection	The **MemWait** input may be connected to one of the strobes S2, S3, S4 by entering 'S2', 'S3' or 'S4' respectively. Alternatively, by specifying a number in the range 1 to 60 **MemWait** may be connected to a simulated external wait state generator. This causes **MemWait** to be held high then to become inactive (low) a set number of **Tm** periods after the start of T2. **Note**: that this mode is not supported directly by the T414; in a final design, a circuit would have to be built to perform this function. If the current connection of **MemWait** causes the signal to become inactive just as **ProcClockOut** is falling during T4, a warning is given that there is a hazard of a **wait race** condition. This is because **MemWait** is sampled on the falling edge of **ProcClockOut** –and if the signal is changing while being sampled, the result is undefined.
EMI clock period Tm	The value of **Tm** for a **clockIn** frequency of 5MHz. This is computed from the other parameters and displayed.

Wait states	The number of wait states in the current configuration. This is computed from the other parameters and displayed.
Standard configuration	The parameters can all be reset to those for one of the built in configurations. There are 13 standard configurations available for the T414, valid configuration numbers being 0 to 11 and 31. For the T400, T425, T800 and the T805 there are 17 standard configurations available, valid configuration numbers being 0 to 15 and 31. If the user selects, for a T414, one of the four configurations which are not available, a message will be displayed indicating that this configuration may not be hardwired on a T414. If the currently set configuration happens to correspond exactly to one of the preset configurations, the tool reports the fact.

Table 6.4 `iemit` page 1 parameters

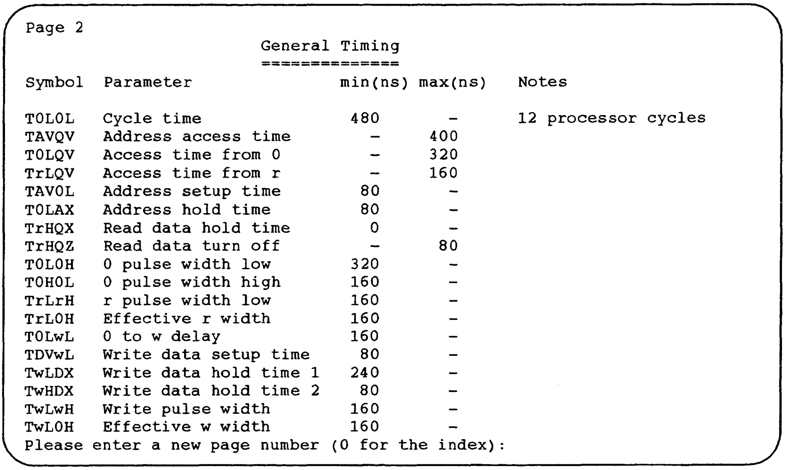
6.4.3 Page 2

This page shows general timing information for the interface, such as delays between various strobes and required access times of the memory devices to be used. The user should adjust these figures to allow for delays in external logic. Table 6.5 lists the timing information displayed on this page while an example of the display is given in figure 6.3.

JEDEC symbol	Parameter description
T0L0L	Cycle time (in both nanoseconds and processor cycles)
TAVQV	Address access time
T0LQV	Access time from **notMemS0**
TrLQV	Access time from **notMemRd**
TAV0L	Address setup time
T0LAX	Address hold time
TrHQX	Read data hold time
TrHQZ	Read data turn off
T0L0H	**notMemS0** pulse width low
T0H0L	**notMemS0** pulse width high
TrLrH	**notMemRd** pulse width low
TrL0H	Effective **notMemRd** width
T0LwL	**notMemS0** to **notMemWrB** delay
TDVwL	Write data setup time
TwLDX	Write data hold time 1
TwHDX	Write data hold time 2
TwLwH	Write pulse width
TwL0H	Effective **notMemWrB** width

Table 6.5 General timing parameters

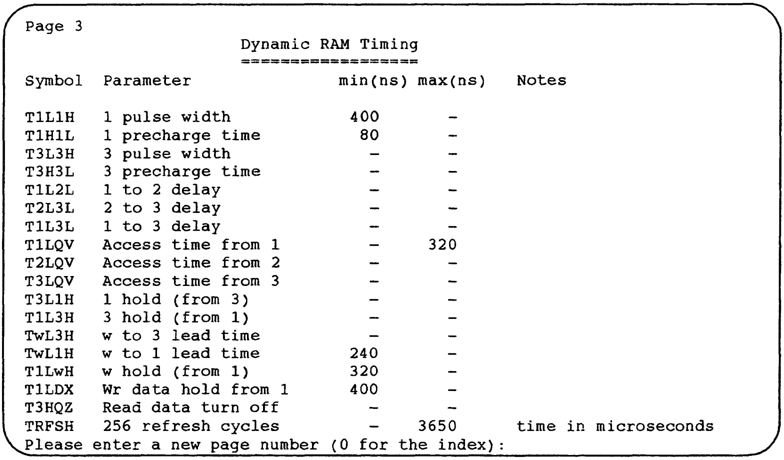
The total cycle time is given in nanoseconds and in processor clock cycles. The only option available from this page is to select another page for display.



```
Page 2
                               General Timing
                               ==============
Symbol  Parameter                min(ns) max(ns)   Notes

T0L0L   Cycle time                480       -      12 processor cycles
TAVQV   Address access time         -     400
T0LQV   Access time from 0          -     320
TrLQV   Access time from r          -     160
TAV0L   Address setup time         80       -
T0LAX   Address hold time          80       -
TrHQX   Read data hold time         0       -
TrHQZ   Read data turn off          -      80
T0L0H   0 pulse width low         320       -
T0H0L   0 pulse width high        160       -
TrLrH   r pulse width low         160       -
TrL0H   Effective r width         160       -
T0LwL   0 to w delay              160       -
TDVwL   Write data setup time      80       -
TwLDX   Write data hold time 1    240       -
TwHDX   Write data hold time 2     80       -
TwLwH   Write pulse width         160       -
TwL0H   Effective w width         160       -
Please enter a new page number (0 for the index):
```

Figure 6.3 Example iemit display page 2

6.4.4 Page 3



```
Page 3
                             Dynamic RAM Timing
                             ==================
Symbol  Parameter                min(ns) max(ns)   Notes

T1L1H   1 pulse width             400       -
T1H1L   1 precharge time           80       -
T3L3H   3 pulse width               -       -
T3H3L   3 precharge time            -       -
T1L2L   1 to 2 delay                -       -
T2L3L   2 to 3 delay                -       -
T1L3L   1 to 3 delay                -       -
T1LQV   Access time from 1          -     320
T2LQV   Access time from 2          -       -
T3LQV   Access time from 3          -       -
T3L1H   1 hold (from 3)             -       -
T1L3H   3 hold (from 1)             -       -
TwL3H   w to 3 lead time            -       -
TwL1H   w to 1 lead time          240       -
T1LwH   w hold (from 1)           320       -
T1LDX   Wr data hold from 1       400       -
T3HQZ   Read data turn off          -       -
TRFSH   256 refresh cycles          -    3650      time in microseconds
Please enter a new page number (0 for the index):
```

Figure 6.4 Example iemit display page 3

This page gives timing information of special interest to designers working with dynamic memory, including various access times and the time for 256 refresh cycles. With this information the designer can ensure that the requirements of the memory devices to be used are met. The user should adjust these figures to allow for delays in external logic. Table 6.6 lists the DRAM timing parameters.

The only option available from this page is to select another page for display. An example of the display is given in figure 6.4.

JEDEC symbol	Parameter description
T1L1H	**notMemS1** pulse width
T1H1L	**notMemS1** precharge time
T3L3H	**notMemS3** pulse width
T3H3L	**notMemS3** precharge time
T1L2L	**notMemS1** to **notMemS2** delay
T2L3L	**notMemS2** to **notMemS3** delay
T1L3L	**notMemS1** to **notMemS3** delay
T1LQV	Access time from **notMemS1**
T2LQV	Access time from **notMemS2**
T3LQV	Access time from **notMemS3**
T3L1H	**notMemS1** hold (from **notMemS3**)
T1L3H	**notMemS3** hold (from **notMemS1**)
TwL3H	**notMemWrB** to **notMemS3** lead time
TwL1H	**notMemWrB** to **notMemS1** lead time
T1LwH	**notMemWrB** hold (from **notMemS1**)
T1LDX	Write data hold from **notMemS1**
T3HQZ	Read data turn off
TRFSH	Time for 256 refresh cycles (in microseconds)

Table 6.6 DRAM timing parameters

6.4.5 Page 4

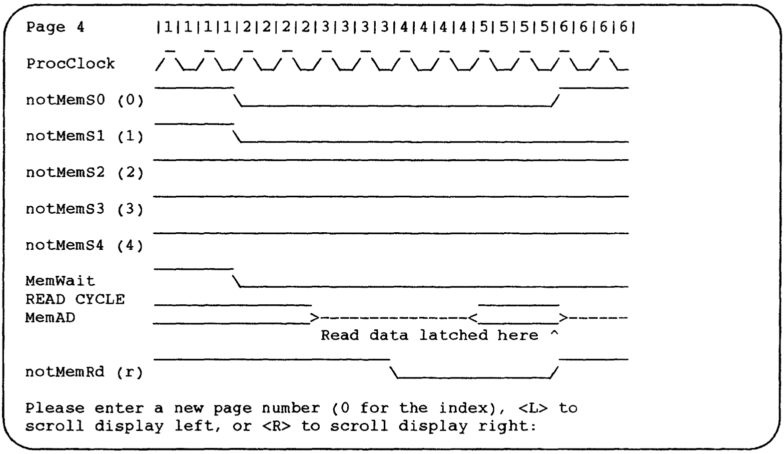
This page shows graphically the timing for a memory read cycle. An example of the display page is given in figure 6.5.

The Tstates and strobes are labelled, and bus activity is shown. The point where data are latched into the processor is also indicated.

At the top of the page is displayed the processor clock and the Tstates, a number indicating the Tstate, 'W' indicating a wait state, and 'E' indicating a state that is inserted to ensure that T1 starts on a rising edge of the processor clock.

Below this are displayed the waveforms of the programmable strobes and the read, write and address/data strobes. Each of these strobes is labelled with the corresponding label parameter.

The point at which the read data is latched is indicated by a '^' beneath the read cycle address/data strobe.



```
Page 4       |1|1|1|1|2|2|2|2|3|3|3|3|4|4|4|4|5|5|5|5|6|6|6|6|
              _   _   _   _   _   _   _   _   _   _   _   _
ProcClock    / \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_
             ________                                  _______
notMemS0 (0)         \________________________________/
             ________
notMemS1 (1)         \_______________________________________
             ________________________________________________
notMemS2 (2)
             ________________________________________________
notMemS3 (3)
             ________________________________________________
notMemS4 (4)
             ________
MemWait              \_______________________________________
READ CYCLE   ________________                 ________
MemAD        ________________>---------------<________>------
                              Read data latched here ^
             ________________________                 _______
notMemRd (r)                         \_______________/

Please enter a new page number (0 for the index), <L> to
scroll display left, or <R> to scroll display right:
```

Figure 6.5   Example iemit display page 4

The **MemWait** waveform shows the input to the **MemWait** pin. If the wait input is a number then it goes low *n* Tm periods after the end of T1 and high again at the end of T6, if the wait input is connected to a strobe it goes low and then high when that strobe does so.

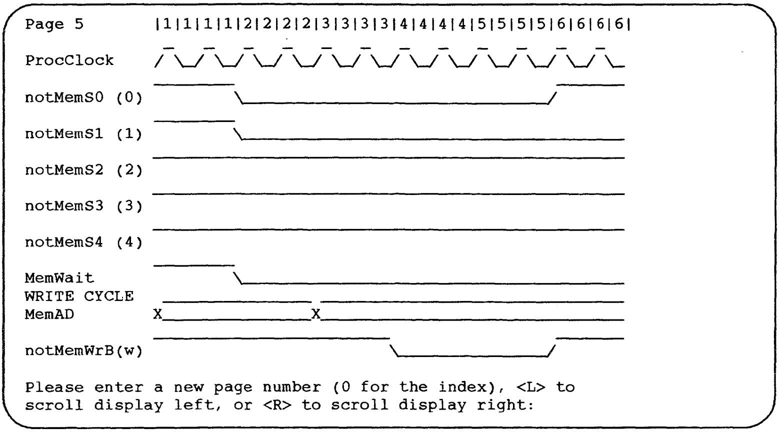
If the cycle is too long to fit horizontally on the screen, it may be scrolled left and right using the L and R keys. The displayed area moves by about 15 characters each time these are used.

6.4.6 Page 5

Page 5 shows the waveforms for a memory write cycle. The display is similar to that of page 4, indeed the read and write cycle diagrams are combined when the PostScript output is produced.

Scrolling the display to the left or right is done in the same way as for page 4.

An example of the display page is given in figure 6.6.



```
Page 5        |1|1|1|1|2|2|2|2|3|3|3|3|4|4|4|4|5|5|5|5|6|6|6|6|
               _   _   _   _   _   _   _   _   _   _   _   _
ProcClock     / \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_
              ________                                ________
notMemS0 (0)          \_______________________________/
              ________
notMemS1 (1)          \_______________________________________
              ________________________________________________
notMemS2 (2)
              ________________________________________________
notMemS3 (3)
              ________________________________________________
notMemS4 (4)
              ________
MemWait               \_______________________________________
WRITE CYCLE    _______________ _______________________________
MemAD         X_______________X_______________________________
              ________________________                ________
notMemWrB(w)                          \_______________/

Please enter a new page number (0 for the index), <L> to
scroll display left, or <R> to scroll display right:
```

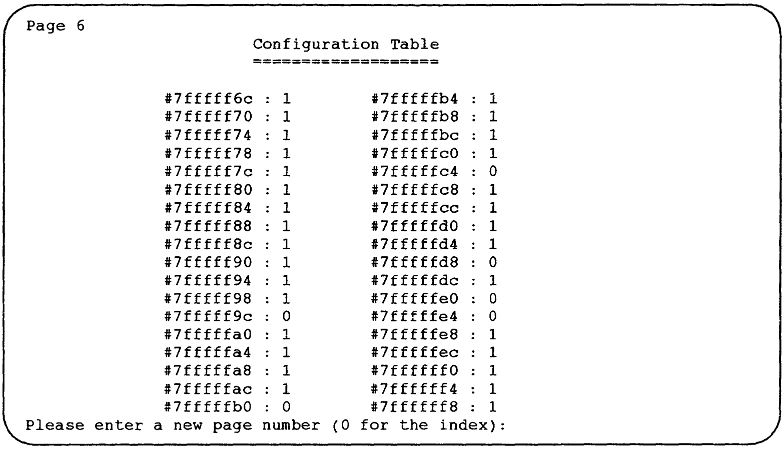
Figure 6.6 Example **iemit** display page 5

6.4.7 Page 6

This page gives a **configuration table** for the current configuration. This is a listing of the data which have to be placed in a ROM situated at the top of the transputer's memory map in order to achieve the desired configuration. The table consists of 36 words of data, but only the least significant bit in each is used. The address and contents are given for each location. **Note**: when `iemit` is used to generate the memory configuration, the **Memconfig** pin must be connected to **MemnotWrD0.**

An example of the display page is given in figure 6.7.

Note: that if page 1 indicates that the configuration is one of the transputer's preset ones, there will be no need for a ROM; configuration can be achieved by connecting the **MemConfig** pin of the device to one of the address/data lines.



```
Page 6
                          Configuration Table
                          ===================

                #7fffff6c : 1          #7fffffb4 : 1
                #7fffff70 : 1          #7fffffb8 : 1
                #7fffff74 : 1          #7fffffbc : 1
                #7fffff78 : 1          #7fffffc0 : 1
                #7fffff7c : 1          #7fffffc4 : 0
                #7fffff80 : 1          #7fffffc8 : 1
                #7fffff84 : 1          #7fffffcc : 1
                #7fffff88 : 1          #7fffffd0 : 1
                #7fffff8c : 1          #7fffffd4 : 1
                #7fffff90 : 1          #7fffffd8 : 0
                #7fffff94 : 1          #7fffffdc : 1
                #7fffff98 : 1          #7fffffe0 : 0
                #7fffff9c : 0          #7fffffe4 : 0
                #7fffffa0 : 1          #7fffffe8 : 1
                #7fffffa4 : 1          #7fffffec : 1
                #7fffffa8 : 1          #7ffffff0 : 1
                #7fffffac : 1          #7ffffff4 : 1
                #7fffffb0 : 0          #7ffffff8 : 1
Please enter a new page number (0 for the index):
```

Figure 6.7 Example iemit display page 6

6.5 iemit error and warning messages

The following is a list of error and warning messages the tool can produce:

Command line parsing error

An option has been specified that the tool does not recognize.

Configuration cannot be hardwired on a T414

The transputers which have a configurable memory interface all have (with the exception of the T414) 17 standard memory configurations available to them. The T414 only has a choice of 13 standard configurations. If the standard configurations 12, 13, 14 or 15 are selected for a T414 the above warning message will be displayed against the selection on page 1.

Input out of range

If the value entered for a numeric parameter is outside the range valid for that parameter, an input out of range warning is displayed, the value cleared from the screen and the program waits for a new value.

MemWait connection error

If an attempt is made to connect S1 to the **MemWait** input an error is displayed because it is a meaningless operation.

No input file specified

This indicates that when trying to invoke the tool to produce an output file, the user has not specified a memory configuration file to use as input.

One and only one of options A or P must be specified

This indicates that when trying to produce an output file, the user has not specified whether the output is to be in ASCII or PostScript format.

Unable to open configuration file '*filename*'

This can occur when attempting to load a memory configuration file and indicates that the tool cannot find the specified input file. Check the spelling of the filename and/or that the file is present.

Unable to open output file *'filename'*

An output filename has been specified incorrectly. Check the format of the filename.

Wait race

If one of the programmable strobes is used to extend the memory cycle then the strobe must be taken low an even number of periods Tm after the start of the memory interface cycle. If the strobe is taken low an odd number of periods after the start then a wait race warning will appear. Should this warning appear, it will remain on display on page 1, until the race condition is removed. Further information can be obtained from reference 1, listed at the start of this chapter.

6.6 Memory configuration file

Memory configuration files are text files which may be generated by a standard text editor or by using the memory interface configuration tool `iemit`, see section 6.2.

By convention memory configuration files have the file extension `.mem`. The file consists of a sequence of statements and comments. The following are considered to be comments:

- Blank lines
- Any line whose first significant characters are '—'
- Any portion of a line following '—'.

Comments are ignored by the `ieprom` and `iemit` tools. Statements are all other lines within the file; they may be interspersed with comments.

Individual statements are constructed of the statement and an associated parameter. These must be separated by at least one space or tab but extra spaces may be inserted before, between, or after them for aesthetic purposes. An example memory configuration file is shown in figure 6.8.

```
-- -----
--
--      Memory configuration file produced
--      by a save command from IEMIT.
--      on Thu Feb 13 15:04:04 1992
--
-- -----

device.type      := T425-25

t1.duration      := 4
t2.duration      := 4
t3.duration      := 4
t4.duration      := 4
t5.duration      := 4
t6.duration      := 4

s0.label         := notMemS0
s1.label         := notMemS1
s2.label         := notMemS2
s3.label         := notMemS3
s4.label         := notMemS4
rs.label         := notMemRd
ws.label         := notMemWrB

s1.duration      := 30
s2.duration      := 30
s3.duration      := 30
s4.duration      := 18

refresh.period   := 72
write.mode       := LATE
wait.connection  := 0
```

Figure 6.8 Example memory configuration file

The statements defined are listed along with their parameters in table 6.7. Further information about specifying parameters is given in section 6.4.2.

Option	Description
`standard.configuration`	0 to 13, or 31 for T414 processors. 0 to 15, or 31 for T400, T425, T800 and T805 processors.
`device.type`	One of the following devices: T400–20 T800–17 T414–15 T800–20 T414–17 T800–22 T414–20 T800–25 T425–17 T800–30 T425–20 T800–35 T425–25 T800–25 T425–30 T805–30

`t1.duration,`
`t2.duration,`
`t3.duration,`
`t4.duration,`
`t5.duration,`
`t6.duration`
	1 to 4 Tm periods. (2 Tm periods = 1 clock cycle). Defines the length in Tm periods of Tstates, T1 to T6, of the memory cycle.

`s0.label,`
`s1.label,`
`s2.label,`
`s3.label,`
`s4.label`
	Each of these parameters accepts two text strings. They are the long (up to 9 characters) and short (1 character) names of the strobes **notMemS0** to **notMemS4.** The names should not contain embedded spaces. Names longer than the permitted number of characters will be truncated.

`rs.label`	As above, the long and short names for the read strobe **notMemRd.**
`ws.label`	As above, the long and short names for the read strobe **notMemWrB**.
`s1.duration`	0 to 31 Tm periods. The S1 strobe goes low at the start of Tstate 2. This parameters defines the number of Tm periods before it goes high.

`s2.duration,`
`s3.duration,`
`s4.duration`
	0 to 31 Tm periods. The S2 to S4 strobes all go high at the end of Tstate 5. These parameters define the number of Tm periods before each strobe goes low.

`refresh.period`	18, 36, 54, 72 or the string "Disabled". This parameter defines the period between refresh cycles as a count of **ClockIn** cycles.
`write.mode`	String value either: "Early" or "Late". Defines the write mode.
`wait.connection`	S2, S3, S4 or a value in the range 0 to 60. This parameter connects **MemWait** to one of the strobes S2, S3, S4 or to simulated external wait state generator.

Table 6.7 Memory Configuration file statements

7 ieprom - ROM program convertor

This chapter describes the EPROM hex tool `ieprom`. This tool is used to convert a ROM-bootable file into one or more files suitable for programming an EPROM.

The chapter describes how to invoke `ieprom` and gives details of the command line syntax. It describes the control file which the tool accepts as input and provides background information on the layout of the code in the EPROM. A description of the various file formats which may be output by the tool is given, including block mode where the output is split up over a number of files. The chapter ends with a list of error messages which may be generated by the tool.

7.1 Introduction

The SGS-THOMSON EPROM software is designed so that programs which have been developed and tested as boot-from-link programs, using the toolset may be placed in ROM with only minor modification (see below).

This has the advantages that an application need not be committed to ROM until it is fully debugged and the actual production of the ROMs can be done relatively late in the development cycle without the fear of introducing new problems.

If a network of transputers is being used, only the root transputer needs to be booted from ROM; once this has been booted it will boot its neighbors by link.

Figure 7.1 shows how a network of five transputers would be loaded from a ROM accessed by the root transputer.

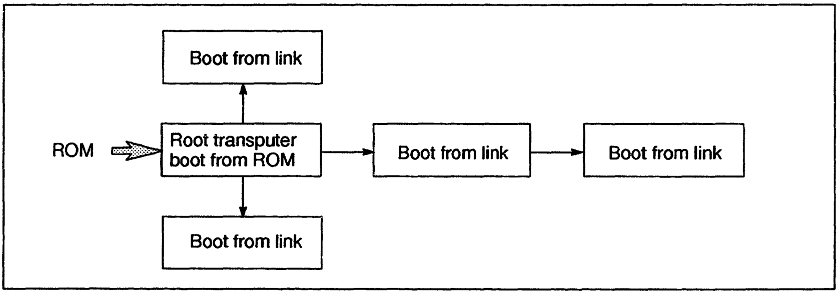


Figure 7.1 Loading a network from ROM

Some 32 bit transputers have a configurable external memory interface. For these transputers a memory configuration file may be created and put into ROM together with the application. A description of memory configuration files and how to create them is given in Chapter 6 (T4/T8-series) and Chapter 14 (ST20 and T450).

7.2 Prerequisites to using the `ieprom` tool

For an application file to be suitable for programming into ROM it must have been configured to be booted from ROM rather than booted from link. This selection is made by specifying the appropriate command line option when using the configurer and collector tools (see the relevant chapters of this manual). It is also essential that all C programs, including those targeted at a single processor, are configured. C programs prepared with the `icollect` '`T`' option are not in a format suitable for `ieprom`.

7.3 Running `ieprom`

`ieprom` takes as input a control file and outputs one or more files which may be put into ROM by an EPROM programmer.

The control file, in text format, specifies the root transputer type, the name of the bootable file containing the application, the memory configuration file (if one is being used), the amount of space available in the EPROM, and the format that the output is to take. Available output formats are: binary, hex dump, Intel, Extended Intel, or Motorola S-Record format.

The `ieprom` tool is invoked by the following command line:

► `ieprom` *filename* { *options* }

where: *filename* is the name of the control file.

options is a list of options from Table 7.1.

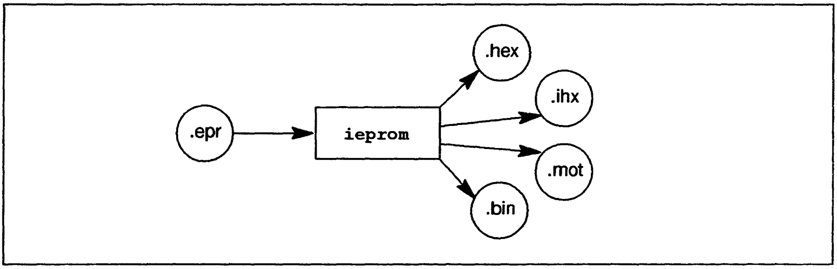
> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.
>
> Options may be supplied in an indirect argument file, prefixed by '@'. See section A.1.2 for details.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
`I`	Selects verbose mode. In this mode the user will receive status information about what the tool is doing during its operation, for example reading or writing to a file.
`R`	Directs `ieprom` to display the absolute address of the code reference point. This address can be used to locate within the memory map created by the `icollect` '`P`' option.

Table 7.1 `ieprom` command line options

The operation of `ieprom` in terms of standard file extensions is shown below.



7.3.1 Examples of use

`ieprom` may be invoked in verbose mode by using the following command:

```
ieprom -i mycontrol.epr
```

7.4 ieprom control file

The control file is a standard text file, prepared with an editor; it consists of comments and statements. A comment is any blank line or any text following the comment marker '—'. Comments are ignored by the `ieprom` tool.

Statements are all other lines within the file. They may be in any order, except that the four statements defining a block must immediately follow the statement '`output.block`' (see table 7.3). Statements may be interspersed with comments.

Individual statements are constructed of a keyword and an associated parameter. These must be separated by at least one space or tab but extra spaces may be inserted before, between, or after them for aesthetic purposes. The statements are listed, along with their parameters, in tables 7.2 to 7.4.

Examples of control file contents are given in section 7.8.

The statements in table 7.2 are used to specify the contents of the EPROM: the processor type, the source of the data (code and memory configuration) to be placed in the EPROM, and the total size of EPROM memory.

Statement	Parameter/Description
`root.processor.type`	*type* This statement specifies the processor type. The processor type can be specified in full (e.g. `T400`), or one of the following classes can be specified: `T2`: 16 bit processor (M212, T212, T222, T225) `T4`: 32 bit processor (T400, T414, T425) `T8`: 32 bit processor with FPU (T800, T801, T805) `T450`: 32 bit processor (ST20450) See appendix B for a full list of valid processor types. This statement *must* be present as the first line in the control file.
`bootable.file`	*filename* This statement specifies the file that contains the output of `icollect`, usually the application plus its ROM loader(s). This statement *must* be present in the control file.
`memory.configuration`	*filename* This statement specifies a T4/T8 memory configuration file to be included in the EPROM image. This file is a standard memory configuration description (see chapter 6 for details). This statement is optional. If absent from the control file then no memory configuration will be inserted in the output file.
`eprom.space`	*hex number* This statement specifies the size of the EPROM memory space in bytes. This space may actually contain several physical devices. This statement *must* be present in the control file.

Table 7.2 Specifying the EPROM contents

The statements in table 7.3 specify the output to be produced: the format of the data and whether the data is to be placed in a single file or split into blocks.

Statement	Parameter/Description
output.format	hex | intel | extintel | srecord | binary This statement specifies the output file format as being one of: plain ASCII hex, Intel hex, extended Intel hex, Motorola S-record or binary format respectively. These output formats are explained in section 7.6. This statement is optional. If absent from the control file then the default output is hex.
output.all output.block	*filename* *filename* These statements are used to specify the type of output and the output filename. By convention the following file extensions should be used: .hex Hexadecimal .bin Binary .ihx Intel formats .mot Motorola format output.all means that all of the image is to be output to one file. output.block specifies that a block of data is to be output to the specified file. It must be followed by the four statements that define the block; these are detailed in table 7.4. The control file *must* contain one output.all statement, or one or more output.block statements.

Table 7.3 Specifying the output format

Table 7.4 lists the statements used to define each output block. One of each of these statements must follow each output.block statement.

Statement	Parameter/Description
`start.offset`	*hex number* This statement specifies the address of the start of the block, as a byte offset into the EPROM space.
`end.offset`	*hex number* This statement specifies the address of the end of the block, as a byte offset into the EPROM space.
`byte.select`	*byte list* | `all` This statement is followed by either a list of byte numbers (separated by `&`), or the keyword `all`. It specifies which bytes in a word are to be output in this block. The byte numbers can be 0, 1, 2 and 3 for 32 bit processors; or 0 and 1 for 16 bit processors.
`output.address`	*hex number* This statement specifies the byte address, in the EPROM programmer's memory map, at which the block is to be output.

Table 7.4 Output block specification

7.5 What goes into the EPROM

This section describes the contents of the EPROM, the reasons behind the code layout and the function of those components inserted by `ieprom`.

The contents of the EPROM includes the bootable file, traceback data and jump instructions to enable the processor to find the start of the bootable file. Should the user define the memory configuration this information will also be placed in the EPROM. The general layout of the code in the EPROM is shown in figure 7.2.

7.5.1 Memory configuration data

Memory configuration data, when present, is placed immediately below the top word of the EPROM. The top word holds the first instructions to be executed if the transputer is booting from ROM.

If the processor has a configurable memory interface it will scan the memory configuration data held on the EPROM, before executing the first instructions. If a standard memory configuration is being used there should be no memory configuration data present and the processor will ignore this section of the EPROM.

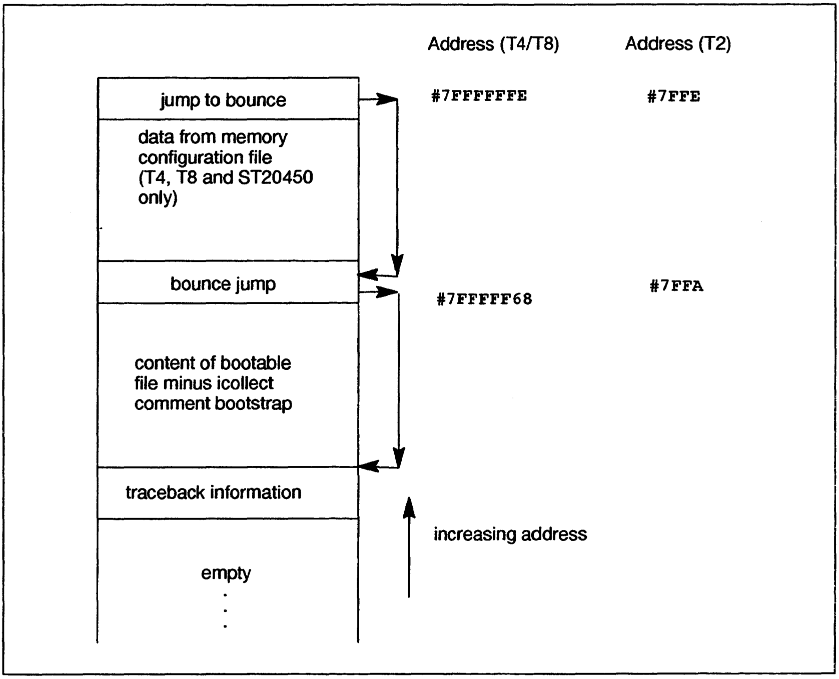


Figure 7.2 Layout of code in EPROM

7.5.2 Parity registers

The T426 has the **ParityErrorReg** and **ParityErrorAddressReg** mapped into the two words immediately below the memory configuration data (addresses #7FFFFF64 and #7FFFFF68). The EPROM tool needs to know that it must avoid these addresses on the T426 and so the processor type must be given explicitly in the **root.processor.type** statement.

7.5.3 Jump instructions

The first instruction executed by the processor when booting from EPROM, is located at *most positive integer* – 1: this is #7FFFFFFE for 32-bit machines and #7FFE for 16-bit machines. The first two instructions cause a backwards jump to be made, with a distance of up to 256 bytes; however, since most applications are larger than 256 bytes it is necessary for ieprom to insert a 'bounce' jump back to the start of the bootable file.

7.5.4 Bootable file

The bootable file will have been produced by the collector tool `icollect`, using a boot from ROM loader. The comment bootstrap, containing traceback information originally added to this file by `icollect`, is stripped off by `ieprom`.

The bootable file is placed in the EPROM such that the start of the file is placed at the lowest address, with the rest of the file being loaded in increasing address locations. The end of the file is placed immediately below the bounce jump instruction, which points to the start of the bootable file.

7.5.5 Traceback information

`ieprom` creates its own traceback information consisting of the name of the control file and the time at which `ieprom` ran. This information is placed below the start of the bootable file. **Note:** at present this information is not used by any of the tools.

7.6 ieprom output files

The tool can produce output in a form readable by the user or in a form readable by EPROM programming devices. The following formats are supported:

- Binary output
- Hex dump
- Intel hex format
- Intel extended hex format
- Motorola S-record format

Whichever form is used, it is sometimes necessary to output the data in separate blocks. Block mode operation is discussed in section 7.7.

Note: there is no output for unused areas of the EPROM. If the buffer in the EPROM programmer is not initialized before loading the files produced by this program into it, unused areas of the EPROM will be filled with random data. Although the operation of the bootstrap code and loader programs will not be affected by the presence of random data, these areas of the EPROM cannot subsequently be programmed without erasing the whole device.

7.6.1 Binary output

This file is in binary format and simply contains all bytes output. There is no additional information such as address or checksums.

7.6.2 Hex dump

This simple format is intended to be used to check the output from the program. The dump consists of rows of 16 bytes each, prefixed by the address of the first byte of each row. The format contains no characters other than the hexadecimal digits, the space character and newlines.

7.6.3 Intel hex format

This is a commonly used protocol for EPROM programming equipment. A sequence of data records is sent. Each record contains a few bytes of information, a start address and a checksum. In addition, a special record marks the end of a transmission. Since the format only supports 16-bit addresses, any longer addresses will generate an error message. Records produced by this program contain at most 32 bytes each.

7.6.4 Intel extended hex format

This format, also known as Intel 86 format, is similar to Intel hex, but adds another type of record. The new type 02 record is used to specify addresses of more than 16 bits. The type 02 record contains a 16-bit field giving a segment base offset. This value is shifted left four places and added to subsequent addresses. This mimics the operation of the segment registers on the Intel 8086 range of microprocessors. The segment base offset value persists until the next type 02 record occurs. This format therefore allows addresses up to 20 bits in length. Again, longer addresses will generate an error message. The program minimizes the number of type 02 records inserted in its output.

7.6.5 Motorola S-record format

This format is another well known industry standard; it consists of a header record, data records, and finally an image end record. The advantage of this format is that, by the use of different data record types, it can support 16, 24, or 32 bit addresses. This program uses whichever data record type is necessary.

7.7 Block mode

Block mode is a term used to describe the output from `ieprom`, when more than one output file is produced. The user defines how the data is to be split between files using control file statements (see table 7.4).

7.7.1 Memory organization

In order to understand the ideas behind block mode operation it is helpful to understand the way memory is organized in a 16 or 32 bit transputer.

In general, a transputer with a 32 bit data bus will expect to read from memory in 32 bit words; the addresses of these words will be on word boundaries (i.e. the address will always be divisible by 4, the two least significant bits will be 0). EPROM devices, however, are usually 8 bits wide, and so it is necessary to have 4 EPROMs side by side to make up the 32 bit width. These 4 devices are addressed as bytes 0 to 3. The two least significant bits of an address (the 'byte selector') give the byte numbers.

Similarly a 16 bit transputer will expect to read from memory in 16 bit words. The address of each word will always be divisible by 2. The two EPROM devices required to make up the 16 bit width will be addressed as bytes 0 and 1. In this case the least significant bit of an address indicates the byte being accessed.

7.7.2 When to use block mode

Block mode has three uses:

- When the EPROM programmer being used is unable to split the input data into bytes, in order to program separate byte wide devices.
- When the EPROM programmer has insufficient memory to hold the entire image.
- When it is necessary, for some reason, to load the program to a different address in the EPROM programmer to that which it will occupy in the EPROM space.

7.7.3 How to use block mode

When block mode is to be used, the user must first decide on the blocks to be output. For each block an `output.block` statement must be specified in the control file. Each `output.block` statement must be followed by the four statements:

```
start.offset
end.offset
byte.select
output.address
```

`ieprom` will scan the entire image and output those bytes that have an EPROM space address between `start.offset` and `end.offset` and whose byte address matches the `byte.select` value. It will output this data to contiguous addresses starting at `output.address`.

Note: if the image does not occupy all of the EPROM space then there may be some space at `output.address` before the data starts.

7.8 Example control files

7.8.1 Simple output

For this example the application is in the file `bootable.btr`, there is no memory configuration, there is 128 kbytes of EPROM, and the EPROM programmer can take all of the code as one file.

```
-- EPROM description file for example 1
root.processor.type  T4
bootable.file        bootable.btr
eprom.space          20000
output.format        srecord
output.all           image.mot
```

7.8.2 Using block mode

In this example the application is in `embedded.btr`, there is a memory configuration in `fastsram.mem`, there are 16 kbytes of EPROM and the data is to be split into four blocks of 4k EPROMs to be programmed at locations 0000, 1000, 2000, and 3000 in the EPROM programmer's memory.

```
-- EPROM description file example 2

root.processor.type  T8
bootable.file        embedded.btr
memory.configuration fastsram.mem
eprom.space          4000
output.format        intel

output.block        part1.ihx
     start.offset     0000
     end.offset       3FFF
     byte.select      0
     output.address   0000

output.block        part2.ihx
     start.offset     0000
     end.offset       3FFF
     byte.select      1
     output.address   1000

output.block        part3.ihx
     start.offset     0000
     end.offset       3FFF
     byte.select      2
     output.address   2000

output.block        part4.ihx
     start.offset     0000
     end.offset       3FFF
     byte.select      3
     output.address   3000
```

7.9 Error and warning messages

The following is a list of error and warning messages the tool can produce:

Command line parsing error

This indicates that a command line option has been specified that the tool does not recognize.

Control file error

This message will be received whenever an error is found in the format of the control file. A self explanatory message will be appended, giving details of what the tool expects the format to be.

No input file specified

This indicates that when trying to invoke the tool the user has not specified a control file to use as input.

Unable to open bootable file *'filename'*

The bootable file specified in the control file cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open configuration file *'filename'*

The memory configuration file specified in the control file cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open control file *'filename'*

The control file specified cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open output file *'filename'*

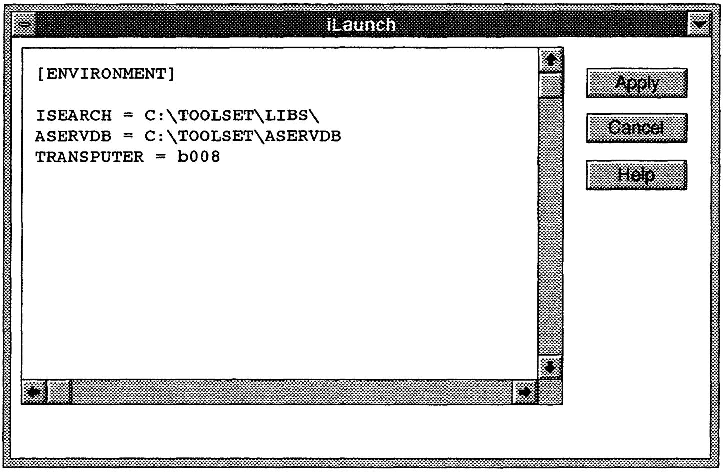
An output filename has been specified incorrectly. Check the format of the filename.

8 `ilaunch` - Windows launch tool

The purpose of the Windows launch tool is to edit (from Microsoft Windows) the contents of the Windows environment file and enable the running of Windows tools from a DOS window.

The application loader `irun` uses the parameters described in Chapter 15. On PC systems using Windows, these parameters may be held either in DOS environment variables or in a Windows environment file. The Windows launch tool `ilaunch` is used to edit the Windows environment file. If `ilaunch` is running, then `irun` and other Windows tools may be run by entering a command in a DOS window. `ilaunch` may only be used with Windows.

The launch tool can be made to start automatically by placing it in the Program Manager's StartUp group; see your Windows User Manual for details. The launch tool will then appear as an icon at the bottom of the screen, and a double click on this icon will open the window shown in Figure 8.1.



```
[ENVIRONMENT]

ISEARCH = C:\TOOLSET\LIBS\
ASERVDB = C:\TOOLSET\ASERVDB
TRANSPUTER = b008
```

Figure 8.1 `ilaunch` window

The text displayed is the content of the Windows environment file. With the `ilaunch` window selected, the text in the Windows environment file may be edited using the mouse, cursor keys and scroll bars.

To apply any edits made, click on the **Apply** button. To revert to the last saved version click on the **Cancel** button.

8.1 The Windows environment file

The Windows environment file is a text file holding the values of certain parameters used by irun and other Windows-based tools. It is divided into sections by section headings in square brackets ([ ]) such as [ENVIRONMENT]. The order of the entries within each section is not significant. The values given in the Windows environment file override any environment variables of the same name and may be overridden by command line options.

The Windows environment file may be used to hold any values used by the tools, including the following values:

- ISEARCH. This is the search path used by some tools. Each directory pathname must be terminated with a backslash (\). Directory pathnames may be separated by semicolons (;) or spaces. Any definition of ISEARCH must appear in the ENVIRONMENT section.
- ASERVDB. This is the pathname of the AServer database file used to define resources, i.e. target connections and host AServer services. Any definition of ASERVDB must appear in the ENVIRONMENT section.
- TRANSPUTER. This is the resource name of the target connection to be used. The resource must be defined in the AServer database. Any definition of TRANSPUTER must appear in the ENVIRONMENT section.
- Values to be passed to the application running on the target. Any such values must appear in the ENVIRONMENT section. Any value defined in the ENVIRONMENT section may be read by an ANSI C application using the getenv function. Values not defined in the ENVIRONMENT section may be read from DOS environment variables defined before Windows was started.

8.1.1 Syntax

The Windows environment file consists of a number of lines. Each line may be an assignment statement, a section heading, a comment or blank.

Section heading

The syntax of a section heading line is:

[*section_name*]

The lines following a section heading up to the next section heading or the end of the file are in the section. All sections are ignored by the tools except the ENVIRONMENT section.

Assignment

The syntax of an assignment statement is:

variable_name = *value*

Only one assignment statement may be given for any one variable.

The contents of the Windows environment file are not case sensitive, so any name may be typed in any combination of upper and lower case. White space (such as space characters and tab characters) between the components of a line are ignored.

Comment

A comment is any line that begins with a semi-colon (;). For example:

```
; This is a comment
```

9 ilibr - librarian

This chapter describes the librarian tool `ilibr` that integrates a group of compiled code files into a single unit that can be referenced by a program. The chapter begins by describing the command line syntax, goes on to describe some aspects of toolset libraries, and ends with some hints about how to build efficient libraries from separate modules.

9.1 Introduction

The librarian builds libraries from one or more separately compiled units supplied as input files. The input files may be any of the following types:

- Compiled object code files produced by the SGS-THOMSON compilers:
  - `oc` (occam 2 compiler),
  - `icc` (ANSI C compiler),
- Library files already generated by `ilibr` (see section 9.2.4).
- Linked object files (see section 9.2.3).

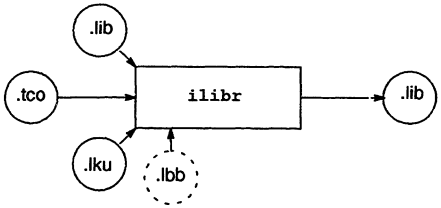
The input files for one library must all be of the same type.

The librarian takes a list of compiled files in TCOFF format and integrates them into a single object file that can be used by a program or program module. Each module in the input list becomes a selectively loadable module in the library. Input files can either be specified as a list on the command line or in *indirect files*.

The library, once built, will contain an index followed by the concatenated modules. The index is generated and sorted by the librarian to facilitate rapid access of the library content by the other tools in the toolset, for example, the linker.

Compiled object files (excluding library files) may be concatenated for convenience before using the librarian. This may prove useful when dealing with a large number of input files.

The operation of the librarian in terms of standard file extensions is shown below.



9.2 Running the librarian

To invoke the librarian use the following command line:

► `ilibr` *filenames* { *options* }

where: *filenames* is a list of input files separated by spaces.

options is a list of one or more options from Table 9.1.

> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options may be supplied in an indirect argument file, prefixed by '@', (see section A.1.2 for details) but must not appear within library indirect files, see section 9.2.2.
>
> Options must be separated by spaces.

The number of file names allowed on a command line is system dependent. To avoid overflow, an indirect file may be used or files (other than library files) may be concatenated. It is the user's responsibility to ensure that the concatenation process does not corrupt the modules, for example by omitting to specify that the concatenation is to be done in binary mode.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
`F` *filename*	Specifies a library indirect file.
`I`	Displays progress information as the library is built.
`O` *filename*	Specifies an output file. If no output file is specified the name is taken from the first input file and a `.lib` extension is added.

Table 9.1 `ilibr` command line options

Example

```
ilibr myprog.t4x myprog.t8x
```

In this example, the compiled object code files `myprog.t4x` and `myprog.t8x` (compiled for T4 and T8 transputers respectively) are used to create a library. Because no output file name is specified on the command line, the library will be given the name `myprog.lib`.

(**Note:** the extensions in this example adopt the `imakef` file naming conventions which indicate transputer target and error mode etc. This is not required but has been done for demonstration purposes. For further details see the appendix covering toolset conventions).

9.2.1 Default command line

A set of default command line options can be defined for the tool using the ILIBRARG environment variable. Options must be specified in the variable using the syntax required by the command line. Options from an environment variable are processed before other options.

9.2.2 Library indirect files

Library indirect files are text files that contain lists of input files, directives to the librarian, and comments. Filenames and directives must appear on different lines. Comments must be preceded by the double dash character sequence '—', which causes the rest of the line to be ignored. By convention indirect files are given the .lbb extension.

Indirect files may be nested within each other, to any level. This is achieved by using the #INCLUDE directive. By convention nested indirect files are also given the extension .lbb.

The following is an example of an indirect file:

```
— user's .lbb file

userproc1.tco             — single modules
userproc2.tco
userproc3.tco
myconcat.tco              — concatenation of modules
#INCLUDE indirect.lbb     — another indirect file
userproc4.tco             — another single module
```

The contents of a nested indirect file will effectively be expanded at the position it occurred.

To specify indirect files on the command line each indirect filename must be preceded by the 'F' option.

9.2.3 Linked object input files

The librarian will also accept linked object files as input, with certain conditions. The facility to create libraries of linked modules provides an easy method of specifying input to the configurer. Such library files should only be referenced from a configuration description.

The librarian will generate an error if an attempt is made to include both linked units and compiled modules in a single library. In addition, libraries of linked object modules must *not* be used as input to the linker ilink. This is because the linker does not accept linked units as input files.

9.2.4 Library files as input

Library files can themselves be used as input files to `ilibr`. When a library file is used as a component of a new library, its index is discarded by `ilibr`.

Library files may not be concatenated for input to the librarian.

9.3 Library modules

Libraries are made up of one or more selectively loadable modules. A module is the smallest unit of a library that can be loaded separately. Modules are selected via the library index.

9.3.1 Selective loading

Libraries can contain the same routines compiled for different transputer types and (for occam modules) in different error modes.

Selection of library modules for linking in with the program is made on the basis of target processor type and error mode. For example, if the program is compiled for an IMS T425 only modules compiled for this processor type or for processors in a compatible transputer class are loaded. For languages such as C the error mode is always UNIVERSAL.

For C modules the linker selects the library modules best suited to the compilation units. For occam the compiler identifies the modules to be selected according to the requirements of the main program. The linker then makes the selection.

The linker also selects library modules for linking on the basis of usage. Only those modules that are actually used by the program are linked into the program.

9.3.2 How the librarian sorts the library index

The librarian creates a library index which is used by the linker to select the required modules. The librarian sorts the index so that for a given processor type, the optimum module is always selected by the linker.

The librarian compares and sorts modules according to a number of factors including attributes set by the compiler options used. These determine for example, the instruction set of the module and influence run-time execution times.

For example, where two library modules were derived from the same source but compiled for classes `TA` and `T8`, the librarian would place the T8 module first because it uses a larger instruction set. Modules compiled with the occam 2 compiler's '`Y`' option are placed earlier in the index than occam modules compiled without the '`Y`' option. This is because the '`Y`' option causes transputer instructions to be used for channel input/output instead of calls to library routines, and thus results in faster code execution. The librarian orders the index entries such that the first valid entry is always the 'best choice'. If two entries are found to be identical the librarian will issue a warning.

9.4 Library usage files

Library usage files describe the dependencies of a library on other libraries or separately compiled code. They consist of a list of separately compiled units or libraries referenced within a particular library. The `.liu` files required by the toolset's libraries are supplied with the toolset.

If the `imakef` tool is used then library usage files should be created for all libraries that are supplied without source. This is to enable the `imakef` tool to generate the necessary commands for linking. Library usage files are text files. They may be created for a specific library by invoking the `imakef` tool and specifying a `.liu` target. See section 12.5.

Such files are given the same name as the library file to which they relate but with an `.liu` extension.

9.5 Building libraries

This section describes the rules that govern the construction of libraries and contains some hints for building and optimizing libraries.

9.5.1 Rules for constructing libraries

1 Routines of the same name in a library must be compiled for different transputer types, error modes or debug attributes.

2 Routines of the same name in a library which will be read by an occam compiler `#USE` directive *must* have the same interface i.e. result type and parameter list, and should have the same functionality.

3 Libraries that contain modules compiled for a transputer class (i.e. TA or TB) are treated as though they contain a copy for each member of the class (using the subset of instructions which are common).

4 Libraries that contain modules compiled in UNIVERSAL mode are treated as though they contain a copy for each of the two error modes HALT and STOP.

5 Libraries that contain occam modules compiled without the '`Y`' option are treated as though they also contain a copy with the '`Y`' option. (When the '`Y`' option is not used for compiling occam code files, channel input/output is performed via library calls, if the '`Y`' option is used then transputer instructions are used for channel input/output).

9.5.2 General hints for building libraries

Routines that are likely to be used together in a program or procedure (such as routines for accessing the file system) can be incorporated into the same library. At a lower level, routines that will *always* be used together (such as those for opening and closing files) can be incorporated into the same module.

Libraries can contain the same routines compiled for different transputer types, in different error modes and with different input/output access to channels. Only those modules actually used by the program are incorporated by the compiler and linked in by the linker.

Where possible compile library input files with debugging enabled. This enables the debugger to locate the library source if an error occurs inside the library.

When building C libraries care should be taken if the 'FS' or 'FC' C compiler command line options are used, that code compatibility is maintained.

9.5.3 Optimizing libraries

It is possible for the user to optimize the size and content of any libraries which he builds himself, to target appropriate processors, improve the speed of code execution and to provide the best code for a given processor. At the same time is can be desirable to build libraries which are flexible i.e. support different transputer targets.

All libraries

Points to consider when constructing libraries in any language or mixture of languages:

- Whether the library is to be targeted at one or two specific processors or a wide range of processors. The transputer type specified for the compilation of a library module determines the instruction set used. Transputer classes TA and TB provide the basic instruction sets common to several transputer types. Transputer classes such as the T5 provide extended instruction sets but are targetted at fewer transputers than classes TA and TB.

- For floating point operations, classes T5 and TB provide better code and therefore better execution times than class TA.

- Whether the versatility of the library should be reduced in order to create a smaller library.

Libraries containing occam modules

When building libraries which include modules written in occam the same considerations apply, but also note the following:

- The error mode used will affect the size of the library. A library created from modules compiled in UNIVERSAL mode will behave as if it contains a copy of the code for both HALT and STOP mode. Also, on the current range of transputers, code compiled in HALT mode will tend to execute faster than if it is compiled in STOP or UNIVERSAL error modes.

- For libraries containing modules where the method of channel input/output may be altered, (such as in occam), both the availability of the interactive debugging facility and the speed at which the code will be executed may be affected.

  By default the compiler will implement channel input/output via library calls. When the occam compiler 'Y' option is used, transputer instructions are used for channel input/output. This leads to faster execution times.

  A side effect of the occam compiler's 'Y' option is that it disables the 'software virtual routing' facilities of the configurer. If the debugging processes require the 'software virtual routing' facilities of the configurer then using the compiler 'Y' option will disable interactive debugging.

  Code which is compiled to use library calls for channel i/o can be called by code compiled to use either library calls or transputer instructions for channel i/o. The opposite case is not true, therefore, in order to build a library which is flexible in this respect, code should be compiled to use library calls.

For a detailed description of transputer types and error modes, see appendix B.

Outlined below are three different approaches to optimization. The first approach provides the greatest level of flexibility in its application. The experienced user may refine these guidelines to specific requirements.

Semi-optimized library build targeted at all processor types

This is the simplest way to build a flexible library that covers the full range of processors supported by this toolset.

The user should compile each module separately for the following four cases and incorporate all four versions into the library.

Processor type/class	Error mode	Method of channel I/O
ST20 or T450	UNIVERSAL	Via library calls
T2	UNIVERSAL	Via library calls
TA	UNIVERSAL	Via library calls.
T8	UNIVERSAL	Via library calls.
Notes: Error mode and channel i/o only apply only to modules which employ them e.g. occam modules compiled by oc.		

The resulting library will be small in terms of the number of modules it will contain. Due to their generic nature the modules themselves may be bulky and because they contain only the base set of instructions, the execution time for the program will tend to be slower than a more optimized approach.

Optimized library targeted at all processor types

In order to build a library which is both generalized enough to work for all 32-bit processors and is then optimized for modules which require extended instructions sets the following approach is recommended:

1 Compile all modules for classes TA, T8 and T450 (or ST20). This will provide modules which can be run on all 32-bit processors.

2 If any of the modules perform floating point operations, compile these modules for class TB as well.

For 16-bit transputers it should be sufficient to compile all modules for class T2.

Library build targeted at specific processor types

This method of building a library will limit the use of the library modules to specific processor types and error modes. It is recommended as the simplest strategy to use when the following options are known for each module:

- Target processor type.
- Error mode of modules, if any, (i.e. HALT, STOP or UNIVERSAL).
- Method of channel input/output, if any.

All modules to be included in the library must be compiled for *each* target processor type required and, if appropriate, for the same error mode and method of channel input/output. The resulting library may be large and contain a certain amount of duplication.

For example, for the following options:

- T425 and T805 processor types
- HALT error mode
- channel input/output via library calls

each module should be compiled for the following:

Processor type/class	**Error mode**	**Method of channel I/O**
`T425`	HALT	Via library calls
`T805`	HALT	Via library calls.
Note: Error mode and channel i/o only apply only to modules which employ them e.g. occam modules compiled by `oc`.		

9.6 Error Messages

This section lists each error and warning message which may be generated by the librarian. Messages are in the standard toolset format which is explained in appendix A.

9.6.1 Information messages

filename - **Previous "Aio failed" message related to file**

An "**Aio failed**"error has been reported. This message identifies the file that was being read/written at the time of the failure.

9.6.2 Warning messages

filename - **bad format: symbol** *symbol* **multiply exported**

An identical symbol has occurred in the same file. There are three possibilities:

The same file has been specified twice.

The file was a library where previous warnings have been ignored.

A module in the file has been incorrectly generated.

filename1 - **bad format: symbol** *symbol* **also exported by** *filename2*

An identical symbol has occurred in more than one module. If the linker requires this symbol, it will never load the second module.

9.6.3 Serious errors

filename - **Cannot mix linked and linkable files**

The librarian is capable of creating libraries from compiled modules or linked units, but it is illegal to attempt to create a library from both.

filename - **Could not open for reading**
filename - **Could not open for writing**

The named file could not be found/opened for reading/writing.

filename - line number - **Unrecognised directive** *directive*

An unrecognized directive has been found in an indirect file.

filename1 - line number - **Could not open** *filename2* **for reading**

The file specified in an `INCLUDE` directive in an indirect file could not be opened.

filename - **bad format: directive outside module**
filename - **bad format: not a TCOFF file**
filename - **bad format: multiple descriptors for symbol**
filename - **bad format: unexpected end of file**
filename - **bad format: unknown expression type**
filename - **bad format: unmatched end module**

These errors are not expected during normal operation. They should only appear if an invalid module is specified to the librarian, or a non-user option has been used incorrectly.

Aio failed: Failed memory allocation
Aio failed: Read failed
Aio failed: Unexpected end of input
Aio failed: Write failed

Failure reported via a low level library routine. These errors will be followed by an *Information* message to identify the file which caused the error.

Command line error *token*

An unrecognized token was found on the command line.

Could not open aio handle

An internal error has occurred which should never be reported during normal operation.

Internal errors should be reported to your local SGS-THOMSON distributor or field applications engineer.

9.6.4 Fatal errors

filename - **Unable to set file buffer**
Aio failed: Invalid abstract i/o handle, NULL or missing installed values
Aio failed: Invalid character in record format
Aio failed: Invalid parameter value
Aio failed: Tell failed
Aio failed: Unexpected read of an 8 byte TCOFF integer

An internal error has occurred which should never be reported during normal operation.

Internal errors should be reported to your local SGS-THOMSON distributor or field applications engineer.

10 ilink - linker

This chapter describes the linker tool **ilink** which combines a number of compiled modules and libraries into a linked object file. The chapter begins with a short introduction to the linker, explains the command line syntax and goes on to describe linker indirect files and the main linker options. The chapter ends with a list of linker messages.

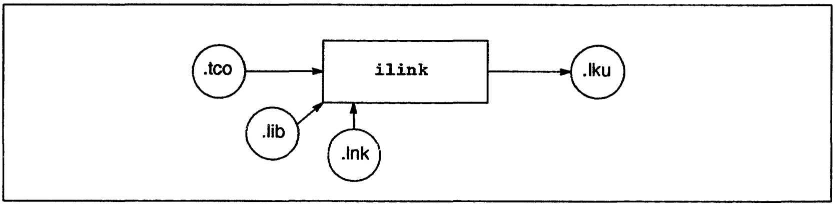
10.1 Introduction

The linker links a number of compiled modules and library files into a single linked object file (known as a linked unit), resolving all external references. The linker may be used to link object files produced by the ANSI C compiler icc and the occam 2 compiler oc. Code produced by the linker can be used as input to the configurer and collector tools to produce a bootable code file.

The linker can be driven directly from the command line or indirectly from a *linker indirect file*. This is a text file which contains a list of files to be linked, together with directives to the linker.

The linker is designed to accept input files in the *T*ransputer *C*ommon *O*bject *F*ile *F*ormat (TCOFF) supported by this release of the toolset. The linker is a 'compacting' linker i.e. it will accept compactable input generated by the compiler and calculate the optimum size of some instructions. This enables the linker to produce smaller and faster code than is possible with non-compactable input, leading to faster execution times for applications. For compatibility with previous toolsets the linker will still accept non-compactable input and generate non-compacted output, although this facility may not be supported by future toolset releases. The linker automatically recognizes whether the input is compactable or non-compactable.

The operation of the linker in terms of standard toolset file extensions is shown below.



10.2 Running the linker

To invoke the linker use the following command line:

▶ `ilink` [*filenames*] { *options* }

where: *filenames* is a list of compiled files or library files.

options is a list of the options given in Table 10.1.

> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options may be supplied in an indirect argument file, prefixed by '@', (see section A.1.2 for details) but must not appear within linker indirect files, see section 10.3.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

If an error occurs during the linking operation no output files are produced.

Examples of use:

```
icc -st20 hello.c
ilink -st20 hello.tco -f cstartup.lnk
icconf hello.cfs
icollect hello.cfb

icc -t450 hello.c
ilink -t450 hello.tco -f cstartup.lnk
icconf hello.cfs
icollect hello.cfb

icc -t805 hello.c
ilink -t805 hello.tco -f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
```

In the above examples, a compiled C file is linked for a specific target, using the standard C startup linker indirect file `cstartup.lnk`. The examples also shows the steps for compiling, configuring and collecting.

Option	Description
Transputer type	See appendix B for a list of options to specify transputer type.
EX	Allows the extraction of modules without linking them. See section 10.5.3.
F *filename*	Specifies a linker indirect file.
H	Generates the linked unit in HALT mode. This is the default mode for the linker and may be omitted for HALT mode programs. This option is mutually exclusive with the 's' option.
I	Displays progress information as the linking proceeds.
KB *memorysize*	Specifies virtual memory required in Kilobytes.
ME *entryname*	Specifies the name of the main entry point of the program and is equivalent to the #mainentry linker directive (See 10.4.4).
MO *filename*	Generates a module information file with the specified name, see section 10.7. **Note:** if this file is to be used as input to imap, it must be given an extension of the form: .dxx. The characters '*xx*'are determined by the 2nd and 3rd characters of the extension given to the linker object file. For example if the linker object file takes the default extension .lku, the information file is given the extension .dku.
NS	Prevents the linker including a symbol table in the linked unit. The linked unit will be smaller, however, the functionality/efficiency of the INQUEST debugging and profiling tools may be impaired if used with the linked unit. See section 10.5.8.
O *filename*	Specifies an output file.
S	Generates the linked unit in STOP mode. This option is mutually exclusive with the 'H' option.
T	Specifies that the output is to be generated in TCOFF format. This format is the default format.
U	Allows unresolved references.
X	Generates the linked unit in UNIVERSAL error mode, which can be mixed with HALT and STOP modes.
Y	Applies only to occam code. Disables the use of library calls for channel input/ output and instead uses transputer instructions. See section 10.5.11.

Table 10.1 ilink command line options

10.2.1 Default command line

A set of default command line options can be defined for the tool using the ILINKARG environment variable. Options must be specified using the syntax required by the command line. Environment options are interpreted before other arguments.

10.2.2 Output format

By default the linker outputs object files in TCOFF format. TCOFF is used by all tools in the toolset. A command line option 'T' has the same effect but is not required.

10.3 Linker indirect files

Linker indirect files are text files containing lists of input files and commands to the linker. Indirect files are specified on the command line using the 'F' option.

10.3 Linker indirect files

Linker indirect files can contain filenames, linker directives, and comments. Filenames and directives must be on separate lines. Comment lines are introduced by the double dash ('—') character sequence and extend to the end of line. Comments must occupy a single line.

Indirect files can include other indirect files.

Linker indirect files must be created for all link operations which involve the use of `imakef` and C modules. For further details see section 12.4.

10.3.1 Linker indirect files supplied with the toolset

Linker indirect files supplied with the toolset are described in the 'Developing programs...' chapter of the '*User Guide*'. The purpose of these files is to reference various runtime libraries (or in the case of occam, compiler libraries) required to link application programs. When specifying the program modules to be linked, the appropriate linker indirect file must be included on the linker command line.

10.3.2 Linking different versions of software after occam upgrade

Because it is possible to specify the full path names of files within linker indirect files they can be used to link different versions of the same occam routines. This may help user's who have used third party libraries which are not supplied in source form, and who wish to upgrade their occam toolset. Problems can occur when the toolset libraries have been upgraded, as they will have the same names but will be different to those used by the third party library. The linker will not link occam libraries with incompatible code.

A possible solution to this situation is to create a linker indirect file which references the third party library and the original copy of the toolset libraries, (specifying their full path name to differentiate them). This can then be linked with the user's program and upgraded toolset libraries.

The following example shows how this could be done for an occam program where the third party library clashes with the upgraded compiler libraries. **Note:** the example uses a UNIX host; path names and the option flags would require changing for other hosts. The user creates a linker indirect file, `lib.lnk`, containing the following:

```
lib.lib                          — third party library
original/libs/occamult.lib       — original compiler
original/libs/occama.lib         — libraries, specifying
original/libs/virtual.lib        — the full path name
```

This is then linked with the user's program and the upgraded compiler libraries, which are linked using a supplied linker indirect file, see section 10.3.1. The name takes the form `occamx.lnk`. For example:

```
ilink —t805 userprog.tco —f occama.lnk —f lib.lnk
```

Note: this is not guaranteed to work. The upgrade may have introduced other incompatibilities and you are normally recommended to recompile after an upgrade.

10.4 Linker directives

The linker supports six directives which can be used to fine tune the linking operation. Linker directives must be incorporated in indirect files (they cannot be specified on the linker command line) and are introduced by the hash ('#') character.

The six linker directives are summarized below and described in detail in the following sections.

Directive	Description
#alias	Defines a set of aliases for a symbol name.
#define	Assigns an integer value to a symbol name. Not applicable to occam programs.
#include	Specifies a linker indirect file.
#mainentry	Defines the program main entry point.
#reference	Creates a reference to a given name.
#section	Defines the linking priority of a module.
Note: Symbol names are case sensitive.	

10.4.1 #alias *basename* {*aliases*}

The #alias directive defines a list of aliases for a given base name. Any reference to the alias is converted to the base name before the name is resolved or defined. For example, if a module contains a call to routine proc_a, which does not exist, then another routine proc_d may be given the alias proc_a in order to force the call to be made to routine proc_d.

```
   #alias proc_d proc_a
```

In the above example the reference to proc_a is considered to be resolved. Modules may be loaded from the library for proc_d but the linker will not attempt to search for library modules for proc_a. If a procedure called proc_a is found in any module then an error will result as the symbol will be multiply defined.

10.4.2 #define *symbolname value*

The #define directive defines a symbol and gives it a value. This value must either be an optionally signed decimal integer, or an unsigned hexadecimal integer. (If it is the latter it must be preceded by a # sign). #define is also discussed in section 10.5.3.

Note: this directive is not applicable to occam.

10.4.3 `#include` *filename*

The `#include` directive allows a further linker indirect file to be specified. Linker indirect files can be nested to any level. The following is an example of nested indirect files:

```
-- user's .lnk file:

userproc1.tco        -- module
#mainentry proc_a    -- main entry point directive
#include sub.lnk     -- nested indirect file

-- user's sub.lnk file:

userproc2.tco        -- further modules
userproc3.tco
userlib.lib           -- library
```

10.4.4 `#mainentry` *symbolname*

The `#mainentry` directive defines the main entry point of the program i.e. the top level function of the program. This directive is equivalent to the '`ME`' command line option. Only one main entry point may be specified. If it is omitted the linker will select the first valid entry point in its input as a default. If there is more than one such symbol the linker will warn that there is an ambiguity.

For C programs the supplied linker indirect startup files define the system main entry point.

10.4.5 `#reference` *symbolname*

The `#reference` directive creates a forward reference to a given symbol. This allows names to be made known to the linker in advance, or forces linking of library modules that would otherwise be ignored. The purpose is to allow the inclusion of library initialization routines which might not otherwise be included. For example:

```
#reference open
```

The above example causes `open` to be included in the link, whether it is needed or not.

10.4.6 `#section` *name*

The `#section` enables the user to define the order in which particular modules occur in the executable code.

In order to use this directive the program modules must have been compiled using the compiler pragma `IMS_linkage` (C programs) or `LINKAGE` (occam programs). Details of the appropriate directive can be found within the compiler reference chapter of this manual.

A compiler directive enables a section name to be associated with the code of a compilation module. A section name may take the default value "pri%text%base" or a name specified by the user.

The linker will place modules associated with the section name "pri%text%base" first in the code of the linked unit, in the order in which these modules are encountered. When the linker directive #section is used this default condition is overridden. The modules identified by user defined section names will be placed first in the linked module, in the order in which the #section directives are encountered. These will be followed by any other modules in an undefined order at the end of the linked unit.

For example:

```
#section first%section%name
#section second%section%name
```

In the above example any modules identified by first%section%name will be linked first, followed by modules identified by second%section%name, followed by any other modules.

10.5 Linker options

10.5.1 Processor types

A number of options are provided to enable the user to specify the target processor for the linked object file, see appendix B. A single target processor or transputer class must be specified and this must be compatible with the processor types or transputer class used to compile the modules.

If any input file in the list is incompatible with the processor type in use, the link fails and an error is reported.

10.5.2 Error modes – options H, S and X

Linked code may be generated in three error modes. For C modules, compiled using icc, the error mode will be UNIVERSAL. occam modules, compiled by oc, may be compiled in one of three error modes as shown in table 10.2.

Error mode	Description
HALT	An error halts the transputer immediately.
STOP	An error stops the process and causes graceful degradation.
UNIVERSAL	Modules compiled in this mode may be run in either HALT or STOP mode depending on which mode is selected at link time.

Table 10.2 Error modes

Modules that are to be linked together must be compiled for compatible error modes. C modules can be mixed with occam modules and occam modules compiled for different error modes may also be mixed. Table 10.3 indicates the compilation error modes which are compatible and the possible error modes they may be linked in.

Compatible error modes	ilink options
HALT, UNIVERSAL	H
STOP, UNIVERSAL	S

Table 10.3 ilink error modes

Note: Modules which have been compiled in UNIVERSAL error mode may be *linked* in this mode using the X option. If the resulting linked unit is then processed by the icollect tool it will be treated as if it had been linked in HALT mode.

The linker will produce an error if an input file is in a mode incompatible with the command line options or defaults. The linker default is to create linked modules in HALT mode unless otherwise specified.

10.5.3 Extraction of library modules – option EX

The EX option instructs the linker to extract the modules which would normally have been linked by the ilink command, and to insert them unmodified into an output file. When the EX option is used, the linker does not produce a linked unit as output. Instead it outputs a concatenation of the component modules that would have made up the linked unit. This file can then itself be used as input to either the linker or librarian. By default the output file produced will have the extension .lku, although it is not a linked unit. An alternative output filename and extension can be specified using the ilink O option.

This mechanism can be used for creating sub units for linking at a later date or for extraction of modules from libraries.

When linking or extracting modules the linker attempts to resolve any unresolved references. The linker U option and the #reference directive are particularly useful for controlling the extraction of unlinked modules. For non–occam modules the #define directive can also be used to refine the selection of modules which are extracted. Linker options and directives used in conjunction with the EX option do not modify the extracted modules, they just influence the selection process.

Example: Extraction from a user library

This example demonstrates how to extract sub–parts of a previously supplied library, which contains modules compiled for an ST20 target.

Suppose we are given a library, mylib.lib, which contains routines with entry points start, run, clear, and stop. These routines may also call other modules which reside in the same library, but we are not concerned about their exact names. We can use the linker's EX option to extract a sub–library, which just contains start, run, and stop, but does not contain clear.

We do this by forcing the linker to 'find' references to start, run and stop, but leave out clear.

1 Create the following linker indirect file x.lnk:

```
-- Items wanted
#reference start
#reference stop
#reference run

-- Libraries
mylib.lib
```

2 Use ilink to extract the required modules and place them in a named file:

```
ilink -st20 -f x.lnk -o sublib.tco -ex
```

This command will create a file called sublib.tco which will contain all the submodules required.

3 The librarian can then be used to create a library:

```
ilibr sublib.tco -o sublib.lib
```

Example: Extraction from a user library, using the run–time library

The example demonstrates how to extract sub–parts of a previously supplied library which uses the run–time library. In this example the library contains modules compiled for a T805 target.

Consider the example described above, but in this case, the routines start, stop and run have calls to the run–time library embedded inside them. We have to tell the linker not to complain about these references, because they will be resolved later, when sublib.lib is used.

1 We do the same as before, but we tell the linker not to complain about unresolved references, by using the U command line flag:

```
ilink -t805 -f x.lnk -o sublib.tco -ex -u
```

2 sublib.tco then be supplied to the librarian in the same way as before.

Example: Extraction from a user library, for multiple processor types

Suppose we are supplied with mylib.lib which contains the routines start, stop, run, and clear for both T400 and TA, and that we wish to create a library sublib.lib which contains everything except clear.

1 We use the same method as the first example to extract the T400 code:

```
ilink -f x.lnk -o sublib.t4 -ex -t400
```

This command will create a file called sublib.t4 which will contain all the submodules compiled for T400.

2 We do the same again for TA:

```
ilink -f x.lnk -o sublib.ta -ex -ta
```

This command will create a file called `sublib.ta` which will contain all the submodules compiled for TA.

3 The librarian can then be used to create a library containing both:

```
ilibr sublib.t4 sublib.ta -o sublib.lib
```

Example: Generation of a completely linkable library

Suppose we have built a library `mylib.lib`, which requires access to the run–time library, and we wish to supply this to another person, without having to supply the run–time library separately. We can arrange for the linker to extract all the required parts of the run–time library and add them to `mylib.lib`.

1 Create a linker indirect file `x.lnk` which contains `#reference` lines for each symbol in `mylib.lib`:

```
-- Items wanted
#reference start
#reference stop
#reference run

-- Libraries
mylib.lib

-- Linker indirect file to access run-time library
#include clibs.lnk
```

The run–time library line should be adjusted depending on the type of processor and toolset which is being used:

Language	When	Linker indirect file	Toolset
C	Full run–time library	`clibs.lnk`	ANSI C and ST20 toolsets
C	Reduced run–time library	`clibsrd.lnk`	ANSI C and ST20 toolsets
occam	T450 and ST20 processors	`occam450.lnk`	occam toolset
occam	32–bit processors without an FPU	`occama.lnk`	occam toolset
occam	32–bit processors with an FPU	`occam8.lnk`	occam toolset
occam	16–bit processors	`occam2.lnk`	occam toolset

2 Use `ilink` to extract the required modules and place them in a named file:

In order to supply modules which are suitable for running on all processors, the extraction should be repeated for each required processor class, e.g.:

```
ilink -t450 -f x.lnk -o sublib.450 -ex

ilink -ta -f x.lnk -o sublib.ta -ex

ilink -t8 -f x.lnk -o sublib.t8 -ex

ilink -t2 -f x.lnk -o sublib.t2 -ex
```

This will create the files sublib.450, sublib.ta, sublib.t8 and sublib.t2 which will contain all the submodules required.

3 The librarian can then be used to create the extended library fulllib.lib which will contain the user library together with any routines which are required from the run–time library.

```
ilibr sublib.450 sublib.ta sublib.t8 sublib.t2 –o fulllib.lib
```

Extraction using #define

A module is the smallest unit the linker can extract from a library, and a module may contain several functions. It is quite likely that a module contains functions which are not required as well as functions which are referenced from modules which are required. To prevent a function from being extracted it is assigned a *dummy* value within a #define directive; any value will do. This causes any reference to it to be satisfied.

When the linker encounters a reference to a required function it will extract the whole module. However, if the module contains a function already specified in a #define directive, the function will be multiply defined and the linker will abort the extraction. It may be wise when a function is not required, to define all functions which are exported from that module, to some dummy value, thereby preventing them all from being extracted.

10.5.4 Display information – option I

This option enables the display of linkage information as the link operation proceeds.

10.5.5 Virtual memory – option KB

The KB option allows the user to specify how much memory the linker will use for storing the image of the users program. By default the linker will attempt to store the entire image in memory. In situations where memory is limited, an amount (≥ 1 Kbytes) may be specified. If the program is larger than the amount specified then the linker will use the host filing system as an intermediate store. A reduction in speed may be expected at link time.

10.5.6 Main entry point – option ME

The ME option defines the main entry point of the program i.e. the point from which linking will start. This option is equivalent to the #mainentry directive and takes as its argument a symbol name which is case sensitive.

Only one main entry point may be specified. If it is omitted the linker will select the first valid entry point in its input as a default. If there is more than one such symbol the linker will warn that there is an ambiguity.

10.5.7 Link map filename – option MO

This option causes a link map file to be produced with the specified name. A file extension should be specified as there is no default available. By convention the first character of the extension should be 'd'; the 2nd and 3rd characters are determined by the extension of the linker object file. For example, if the linker object file takes the default extension `.lku`, the map file should be given the extension `.dku`.

If the MO option is not specified, a separate link map file is not produced.

A link map file is a text file containing information about the position of modules in the code file, see section 10.7. It is intended to be used as input to the `imap` tool, see chapter 13.

10.5.8 Suppress symbol table – NS

The linker will place symbol table information in the linked unit which it produces, unless the 'NS' command line option is specified.

The symbol table contains an entry for each named symbol defined in each object module or library module used in the linked unit. The symbol table may be used by the INQUEST debugger and profiling tools.

10.5.9 Linked unit output file – O

The name of the linked unit output file can be specified using the O option. If the option is not specified the output file is named after the first input file given on the command line and a `.lku` extension is added. If the first file on the command line is an indirect file the output file takes the name of the first file listed in the indirect file.

Note: Because there is no restriction on the order in which files may be listed it is up to the user to ensure that the output file is named appropriately.

10.5.10 Permit unresolved references – option U

The linker normally attempts to resolve all external references in the list of input files and reports any that are unresolved as errors.

Sometimes it is desirable to allow unresolved external references, for example during program development. The U option allows the link to proceed to completion by assuming unresolved references are to be resolved as zero. Warning messages may still be generated and the program will only execute correctly if such references are in fact redundant.

10.5.11 Channel input/output – Y

This option applies only to the occam modules.

By default the compiler will generate calls to library routines to perform channel input and output, rather than using the transputer's instructions. The compiler's 'Y' option forces the compiler to use sequences of transputer instructions for channel input and output, resulting in faster code execution. When modules have been compiled using the 'Y' option they must also be linked using the linker's 'Y' option, in order to maintain compatibility.

Note: that code which is linked to use transputer instructions for channel input/output may call code which uses library routines to perform channel input/output, but *not* vice versa.

10.6 Selective linking of library modules

Library modules that are compiled for incompatible processor types or error modes are ignored by the linker. This allows library modules to be selectively loaded for specific processor types or transputer classes.

Libraries supplied with the toolset are supplied in several forms to cover the complete range of transputer types. User libraries that are likely to be used on different transputer types should be supplied for all transputer types likely to be used.

Libraries are also selected for linking on the basis of previous usage. Modules that are used by several input files are linked in only once.

10.7 The link map file

Module data and details of the target processor are always included in the linked unit output file in the form of a comment. This information may also be directed to a named output file by using the MO command line option.

The file contains a map of the code being linked and contains information which may assist the user during program debugging. It is intended as input to the imap tool, see chapter 13.

The map file is generated in text format and covers two categories of input file; separate compilation units, and library modules. The map consists of single line records containing a number of fields. Fields have a single character name followed by a colon. The following information is included:

10.7.1 MODULE record:

A module record is created for each component module in the linked unit.

Record name	Description
N	Module number assigned by the linker.
S	Source filename, may be empty if string is unobtainable.
F	Object filename, the name of the file of library from which the module has been loaded. This will be the full path name.
O	File offset, the offset (in bytes) of the module within its object file.
R	Reference, an external symbol that is used for loading the module from a library. This field will be blank it the module was not loaded from a library.
M	The compilation mode, processor type/class and error mode.

10.7.2 SECT record:

A section record for each section in the linked unit, shows where it is located.

Record name	Description
N	Section number assigned by the linker.
R	Name of the section.
A	Section attributes, where R – read, W – write, X – execute, D – debug, V – virtual.
P	Whether the code has been placed at a fixed address; either N (no) or Y (yes).
O	The offset in bytes of the section within the code.
S	The size in bytes of the section.

10.7.3 MAP record:

This record shows how a region of the linked unit is mapped to a module and section.

Record name	Description
M	Module number of the module that supplied this region.
R	Section number of the section in which this region lies.
A	Address of the region, in bytes.
S	Size of the region, in bytes.

10.7.4 Value record:

This record shows the value of a symbol after linkage.

Record name	Description
N	Name of the symbol.
O	Name of the origin symbol – occam modules only. (Used by the linker to ensure the order of compilation is correct in respect to #USE) .
M	Module number of the exporting module.
U	Whether the symbol has been used (externally at least); either N (no) or Y (yes).
V	Value of the symbol after linking. Expressed as a decimal integer or as a section number plus byte offset into that section.

10.7.5 LOCALVALUE record

This record shows the value of a local symbol after linkage.

Record name	Description
N	Name of the symbol.
M	Module number of the module defining the symbol.
I	TCOFF identity of the symbol within the module which defines the symbol.
V	Value of the symbol after linking. Expressed as a decimal integer or as a section number plus byte offset into that section.

10.8 Using imakef for version control

The imakef tool may be used to simplify the linking of complex programs, particularly those which use libraries that are nested within other libraries or compilation units.

Note: For imakef to function correctly the special file extension system described in section 12.3 and appendix A *must* be used.

10.9 Error messages

This section lists each error and warning message that can be generated by the linker. Messages are in the standard toolset format which is explained in appendix A.

10.9.1 Warnings

filename - **ambiguous main entry points, choosing** *symbol*

If no main entry point is specified, either explicitly or by linker directive, and no main entry point occurs in the input object files and libraries, then the linker defaults to a main entry point.

The default main entry point chosen, is the first entry point in the first module that is linked. If the first module is an occam module, then it may contain more than one candidate entry point. The first encountered will be chosen.

filename - **bad format:** *reason*

The named file does not conform to a recognized SGS-THOMSON file format or has been corrupted.

filename - symbol, **implementation of channel arrays has changed**

Only generated in programs where occam code is used that was compiled in LFF format. The implementation of channel arrays in occam differs between the earlier occam 2 compiler and the current TCOFF-based configurer, and channel arrays cannot therefore be used as parameters to configured procedures.

filename - **symbol** *symbol* **not found**

The specified symbol was not found in any of the supplied modules or libraries.

file1 - **usage of** *symbol* **out of step with** *file2*

May be generated when linking programs incorporating occam modules with a #USE directive, which causes the compiler to scan the file for details concerning certain program resources. This file *must* be unchanged at link time, and the message indicates that this is not the case. There are several possible causes:

1 *file2* has been recompiled after *file1*, in which case *file1* requires recompiling.

2 The file that occurred in the #USE directive has been replaced by a different version of the file at link time.

3 The file that occurred in the #USE directive has not been supplied to the linker, but the linker has located a different version of a required entry point elsewhere.

The occam compiler oc may need to scan certain libraries, of which the user is unaware. Specifying one of the special occam linker indirect files occam2.lnk, occama.lnk or occam8.lnk should take care of these 'hidden' libraries.

Size **bytes too large for 16 bit target**

The code part of the linked unit has exceeded the address space of the T212 derived processor family.

10.9.2 Errors

filename **- name clash with** *symbol* **from** *filename*

May be generated when linking mixed language programs incorporating occam modules.

In languages such as occam entry points may be scoped, i.e. extra information is associated with each symbol to indicate which version of that entry point it is. This allows programs to be safely linked even though there are several different versions of the same entry point occurring at different lexical levels within the program.

This error indicates that a language without occam-type scoping has been mixed with a scoped language and a name conflict has occurred between a scoped and non scoped symbol.

filename **- symbol** *symbol* **multiply defined**

The symbol, introduced in the specified file, has been introduced previously, causing a conflict. The same module may have been supplied to the linker more than once or there may be two or more modules with the same entry point or data item defined.

filename **- symbol** *symbol* **not found**

The specified symbol was not found in any of the supplied modules or libraries.

filename **- usage of** *symbol* **out of step with** *namefile*

May be generated when linking programs incorporating modules with a #USE directive which causes the compiler to scan the file for details concerning certain program resources. This file *must* be unchanged at link time, and the message indicates that this is not the case. There are several possible causes:

1 *file2* has been recompiled after *file1*, in which case *file1* requires recompiling.

2 The file referenced by the #USE directive has been replaced by a different version of the file at link time.

3 The file referenced by the #USE directive has not been supplied to the linker, but the linker has located a different version of a required entry point elsewhere.

The occam compiler oc may need to scan certain libraries, of which the user is unaware. Specifying one of the special occam linker indirect files occam2.lnk, occama.lnk or occam8.lnk should take care of these 'hidden' libraries.

10.9.3 Serious errors

filename - **bad format:** *reason*

The named file does not conform to a recognized SGS-THOMSON file format or has been corrupted.

filename - line number - **bad format: excessively long line in indirect file**

A line is too long. The length is implementation dependent, and is set to 256 characters in this version.

filename - line number - **bad format: file name missing after directive**

A directive (such as include) has no file name as an argument.

filename - line number - **bad format:** *directive* **invalid number**

A numeric parameter supplied to a directive does not correspond to the appropriate format, i.e. decimal or hexadecimal.

filename - **bad format: multiple main entry points encountered**

A symbol may be defined to be the main entry point of a program, instead, multiple main entry points have been defined. Only one main entry point symbol may exist within a single link.

filename - *linenumber* - **bad format: non ASCII character in indirect file**

The indirect file contains some non printable text. A common mistake is to specify a library or module with the 'F' command line argument or an include directive.

filename - **bad format: not linkable file or library**

The linker expects that all files names presented without a preceding switch (on the command line) or directive (in an indirect file) are either libraries or modules.

filename - line number - **bad format: only single parameter for** *directive*

The directive has been given too many parameters.

Cannot create output without main entry point

No main entry point has been specified.

Command line: 1k minimum for paged memory option

When using the KB option, the amount of memory used to hold the image of the program being linked is specified. There is a minimum size of 1k.

Command line: *token*

An illegal token has been encountered on the command line.

Command line: bad format number

A numerical parameter of the wrong format has been found.

Command line: image limit multiply specified

The command line option 'KB' has been specified more than once.

Command line: 'load and terminate' option set, some arguments invalid

Options to load and terminate the linker have been specified in conjunction with other command line options. The linker cannot execute these options if it has been instructed to terminate first.

Command line: multiple debug modes

The command line option 'Y' has been specified more than once.

Command line: multiple error modes

More than one error mode has been specified to the linker.

Command line: multiple module files specified

The command line option 'MO' has been specified more than once.

Command line: multiple output files specified

The command line option 'O' has been specified more than once.

Command line: multiple target type

More than one target processor type has been specified to the linker.

Command line: only one output format allowed

The options 'T', 'LB' and 'LC' are mutually exclusive.

filename **- could not open for input**

The named file could not be found/opened for reading.

filename **- could not open for output**

The named file could not be opened for writing.

filename - line number - **could not open for reading**

The file name specified in an include directive could not opened.

Could not open temporary file

The 'KB' option has been used in a directory where there is no write access or not enough disc space.

filename - **mode:** *mode* - **linker mode:** *mode*

The linker has been given a module to link which has been compiled with attributes incompatible with the options (or lack thereof) on the linker command line.

Invalid or missing descriptor for main entry point *symbol*

Applies to occam modules only. The specified main entry point to the program does not have a valid occam descriptor. This occurs if the wrong symbol name has been used to specify the main entry point.

Multiple main entry points specified

The main entry point has been specified on the command line or in an indirect file more than once.

filename - line number - *directive* **not enough arguments**

The wrong number of arguments have been supplied to a directive.

filename - **nothing of importance in file**

The file name specified in an include directive was empty or contained nothing but white space or comments.

Nothing to link

Various options have been given to the linker but no modules or libraries.

filename - line number - **only one file name per line**

More than one file name has been placed on a single line within an indirect file.

filename - line number - *directive* **too many arguments**

The wrong number of arguments have been supplied to a directive.

Unknown error modes not supported in the LFF format
Unknown processors not supported in the LFF format

When generating LFF format files, certain constructs will have no representation. For example processor types that have come into existence since the LFF format was defined.

filename - line number - **unrecognised directive** *directive*

An unrecognized directive has been found in a linker indirect file.

10.9.4 Embedded messages

Tools that create modules to be linked with `ilink` may embed "messages" within them. Three levels of severity exist; serious, warning, and message. The documentation of the appropriate tool should be consulted for more information. The format of these messages is as follows:

Serious - `ilink` - *filename* - **message:** *message*
Warning - `ilink` - *filename* - **message:** *message*
Message - `ilink` - *filename* - *message*

11 ilist - binary lister

This chapter describes the binary lister tool ilist, which takes a binary file and displays information about the file in a readable form. The chapter provides examples of display options and ends with a list of error messages which may be generated by ilist.

11.1 Introduction

The binary lister tool ilist reads a binary file, decodes it, and displays useful information it on the screen. The output may be redirected to a file. Command line options control the type of data displayed.

The ilist tool can decode and display files produced by the SGS-THOMSON C and occam 2 compilers, by the linker, librarian, configurer and collector tools. Files in editable ASCII format are listed without processing.

Also, because ilist uses the same method to locate files as the other tools (see section A.4) it can be used to find and display the location of header files and library files on the search path specified by ISEARCH.

11.2 Data displays

There are several categories of data that can be displayed when decoding the output from the compilers, linker or librarian. Categories are selected by options on the command line. The main categories are:

- *Symbol data* – symbol names in each module. Information is displayed in tabular form.
- *External reference data* – names of external symbols used by each module. Information is displayed in tabular form.
- *Module data* – data for each module including target processor, compilation mode, and module file name.
- *Code listing* – code contained in each module, displayed in hexadecimal format.
- *Index data* – the content of library indexes.
- *Procedural data* – for external occam routines only.

Note: that for C programs compiled with the 'NOCOMPACT' option, ilist will produce more detailed information.

11.2.1 Modular displays

Object code files reflect the modular structure of the original source. Single unit compilations produce a file containing a single object module, whereas units containing many compilations, such as libraries and concatenations of modules, produce object files with as many object modules. The data produced by ilist reflects the modular composition of object files.

11.2.2 Example displays used in this chapter

Except where indicated, the example displays used in this chapter show the output generated from the lister for a compiled (`.tco`) file generated by `icc`.

11.3 Running the binary lister

To invoke the binary lister use the following command line:

► `ilist` { *filenames* } { *options* }

where: *filenames* is a list of one or more files to be displayed.

options is a list of one or more of the options given in Table 11.1.

Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

Options may be supplied in an indirect argument file, prefixed by '@', (see section A.1.2 for details)

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
`A`	Displays all the available information on the symbols used within the specified modules.
`C`	Displays the code in the specified file as hexadecimal. This option also invokes the '`T`' option by default.
`E`	Displays all exported names in the specified modules.
`H`	Displays the specified file(s) in hexadecimal format.
`I`	Displays full progress information as the lister runs.
`M`	Displays module data.
`N`	Displays information from the library index.
`O` *filename*	Specifies an output file. If more than one file is specified the last one specified is used.
`P`	Displays any procedural interfaces found in the specified modules.
`R` *reference*	Displays the library module(s) containing the specified reference. This option is used in conjunction with other options to display data for a specific symbol. If more than one library file is specified the last one specified is used.
`T`	Displays a full listing of a file in any file format.
`W`	Causes the lister to identify a file. The filename (including the search path if applicable) is displayed followed by the file type. This is the default option.
`X`	Displays all external references made by the specified modules.

Table 11.1 `ilist` command line options

Note: Options will only be applied to files of the appropriate file type. If the file cannot be displayed by the specified option, an error message is generated and the file is not displayed.

ilist will attempt to identify the file type by its contents. If only filenames are supplied, ilist assumes the default option 'W" and simply displays the file's identity.

Example of use:

```
ilist hello.tco –a
```

Examples of ilist usage and the displays generated by the options can be found in succeeding sections.

11.3.1 Options to use for specific file types

Table 11.2 lists the available options and indicates which file formats they may be used to list. The table also lists the file types it is recommended to use with each option, in order of usefulness.

Option	Permitted file format	Recommended usage
H	Any format	
O	Any format	
T	Any format	
W	Any format	
A	TCOFF only	.lib, .tco, .lku
C	TCOFF only	.tco, .lku, .lib, .nif
E	TCOFF only	.lib, .tco, .lku
M	TCOFF only	.tco, .lku, .lib
N	TCOFF libraries only	.lib
P	TCOFF only	.lib, .tco, .lku
R	TCOFF libraries only	.lib
X	TCOFF only	.lib, .tco, .lku

Table 11.2 Recommended options

11.3.2 Output device

ilist sends its output to the standard output stream on the host, normally the terminal screen. Facilities available on the host system may allow you to redirect the output to a file, or send it to another process, such as a sort program. For details of these facilities consult the documentation for your system.

11.3.3 Default command line

A set of default command line options can be defined for the tool using the ILISTARG environment variable. Options must be specified using the syntax required by the command line. Options in environment variables are processed before other options.

11.4 Specifying an output file – option O

The O option enables the user to redirect the display data to an output file. If more than one output file is specified on the command line then the last one specified is used. File extensions should be specified, because defaults are not assumed.

Display options are described in the following sections 11.5 to 11.15. Options are given in alphabetical order.

11.5 Symbol data – option A

This option displays all the available information about the symbols used within the specified modules. A tabular format is used.

Note: The data produced by this display is extensive and detailed and assumes some knowledge of the object file format.

The following information is given:

- Symbol name.
- Section attributes, if applicable.
- Symbol attributes.
- The number of the symbol within the module plus the number of its origin.
- Module name.
- Target processor.
- Error mode.
- occam 2 compiler's 'Y' option - if used indicated by the presence of a 'Y' character. The compiler's 'Y' option causes transputer instructions to be use for channel input/output. If this field is blank this indicates that library calls are being used for channel input/output.

11.5.1 Specific section attributes

Certain attributes apply only to symbols which are section names. If they are applicable, these attributes are indicated by the following nomenclature and displayed as a character string:

R	Read section.
W	Write section.
X	Execute section.
D	Debug section.
V	Virtual section.

11.5.2 General symbol attributes

Attributes for all symbols, including section names, are also indicated by a character string, using the following nomenclature:

Symbol	Description attribute
L	Symbol local to the module.
E	Symbol exported from the module.
I	Symbol imported to the module.
W	Weak attribute, indicates that the symbol takes the value 0 when not defined.
C	Conditional attribute, indicates that the first value given to the symbol is always used.
U	Unindexed, indicates that the symbol is not present in the library index.
P	Provisional attribute, indicates that the last value given to the symbol is always used.
O	Indicates that the symbol is an origin symbol. The origin symbol is used by the linker to check the origin of the module.

Symbol attributes are displayed immediately after the section attributes, and each attribute is displayed at a specific position in the string. Attributes which are not present are indicated by a hyphen '-'.

The position of each attribute in the string is as follows:

```
RWXDV LEIWCUPO
```

11.5.3 Example symbol data display

Figure 11.1 shows the symbol data display for the compiled file **hello.tco**.

```
main                ----- -E------ 0          hello.c          ST20 X
_IMS_printf         ----- --I----- 1          hello.c          ST20 X
text%base           R-X-- -E------ 2          hello.c          ST20 X
local%text          ----- L------- 3          hello.c          ST20 X
```

Figure 11.1 Example output produced by the A option.

11.6 Code listing – option C

The 'C' option produces a full listing of the code in the same format as that generated by the 'T' option, but with the addition of a hex listing of the code at each LOAD_TEXT directive. This option *may* be accompanied by the ‘T’ option; if the ‘T’ option is not specified it is supplied automatically. **Note:** that when the ‘C’ option is used with Network Initialization files (`.nif`) the output will be in the same format as that produced by the ‘T’ option.

The output from this option gives an ASCII dump, in hexadecimal format, of the code for each module. It can be used on any object code.

When used to display object code, the code for each module is displayed as a contiguous block of lines, where each line has the format:

address ASCII hex ASCII characters

where: *address* is the address of the first byte on the line, expressed as an offset from the start of the module.

ASCII hex is the hex representation of the code.

ASCII characters are the ASCII characters corresponding to the hex code.

In all cases code is read from left to right. If a value is not printable it is replaced by a dot ‘.’.

11.6.1 Example code listing display

Figure 11.2 shows the code listing display for the compiled file **`hello.tco`**.

```
00000000 LINKABLE
00000002 START_MODULE CORE FMUL FPSUP DUP WSUBDB MOVE2D CRC BITOPS FPTSTERR
 LDDEVID LDMSTVL POP RMC_CORE1 RMC_CORE2 RMC_CORE3 SEMAPHORE DEVICE BIT32 M
S<=28 ICALL X lang: ANSI_C ""
00000010 VERSION tool: icc origin: hello.c
0000001E SYMBOL EXP ROU "main" id: 0
00000028 SYMBOL IMP ROU "_IMS_printf" id: 1
00000039 LOCAL_SYMBOLS number: 2 ids: 2 to 3
0000003C SECTION REA EXE EXP "text%base" id: 4
0000004A SET_LOAD_POINT id: 4
0000004D SYMBOL LOC "local%text" id: 5
0000005B DEFINE_LABEL id: 5
0000005E COMMENT PRINT "INMOS C compiler  Version  4.01.09 (02:24:52 Mar 11
 1995)  (SunOS-Sun4)"
000000AA DEFINE_LABEL id: 0
000000AD LOAD_PREFIX size: 0 SV:2-SV:3 instr: ldc
000000B6 LOAD_TEXT bytes: 2
000000B9 21FB                                        !.
000000BB DEFINE_LABEL id: 3
000000BE LOAD_TEXT bytes: 1
000000C1 71                                          q
000000C2 LOAD_PREFIX size: 0 AP(SV:1-LP) instr: call
000000CB LOAD_TEXT bytes: 3
000000CE 4022F0                                      @".
000000D1 ALIGN modulo: 4
000000D4 DEFINE_LABEL id: 2
000000D7 LOAD_TEXT bytes: 16
000000DA 0A48656C 6C6F2057 6F726C64 0A002020      .Hello World..
000000EA COMMENT bytes: 5
000000F4 COMMENT bytes: 37
0000011E END_MODULE
```

Figure 11.2 Example output produced by the C option

11.7 Exported names – option E

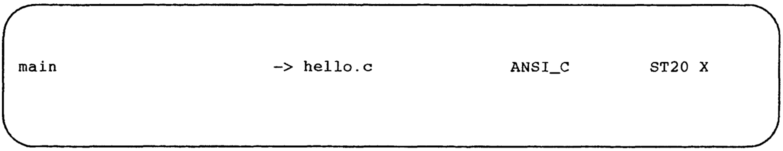
The output from this option is in a tabular format. It consists of a list of names exported by the modules. This option also displays any globally visible data.

The following information is given by the display:

- Exported name.
- The name of the module in which the exported name is found.
- Language used.
- Target processor.
- Error mode.
- occam 2 compiler's 'Y' option - if used indicated by the presence of a 'Y' character. The compiler's 'Y' option causes transputer instructions to be use for channel input/output. If this field is blank this indicates that library calls are being used for channel input/output.

11.7.1 Example exported names display

Figure 11.3 shows the exported names display for the compiled file hello.tco.



```
main                         -> hello.c            ANSI_C         ST20 X
```

Figure 11.3 Example output produced by the E option

11.8 Hexadecimal/ASCII dump – option H

This option provides a display of the specified files in hexadecimal and ASCII format. The option does not attempt to identify file types and may be used to display any files which the lister has previously identified incorrectly.

The output takes the form of a hexadecimal representation of the whole of the file content. The display has a similar appearance to that produced by the C option, however, the C option only functions on *code* found within the file.

Each file is displayed as a contiguous block of lines, where each line has the format:

address ASCII hex ASCII characters

where: *address* is the address of the first byte on the line, expressed as an offset from the start of the file.

ASCII hex is the hex representation of the code.

ASCII characters are the ASCII characters corresponding to the hex code.

In all cases code is read from left to right. If a value is not printable it is replaced by a dot '.'.

11.8.1 Example hex dump display

Figure 11.4 shows the hex dump display for the compiled file **hello.tco**.

```
00000000 0100020C FDFF661E 0EFD42E0 07000400     ......f...B.....
00000010 1B0C0369 63630768 656C6C6F 2E631E08     ...icc.hello.c..
00000020 FC020104 6D61696E 1E0FFC04 010B5F49     ....main......_I
00000030 4D535F70 72696E74 660D0102 0B0C0602     MS_printf.......
00000040 09746578 74256261 73650401 041E0C01     .text%base......
00000050 0A6C6F63 616C2574 6578740E 0105144A     .local%text....J
00000060 00014749 4E4D4F53 20432063 6F6D7069     ..GINMOS C compi
00000070 6C657220 20566572 73696F6E 2020342E     ler  Version  4.
00000080 30312E30 39202830 323A3234 3A353220     01.09 (02:24:52
00000090 4D617220 31312031 39393529 20202853     Mar 11 1995)  (S
000000A0 756E4F53 2D53756E 34290E01 00070700     unOS-Sun4)......
000000B0 07030203 03040603 0221FB0E 01030602     .........!......
000000C0 01710707 000D0703 01020906 04034022     .q............@"
000000D0 F00A0104 0E010206 11100A48 656C6C6F     ...........Hello
000000E0 20576F72 6C640A00 20201408 0000050D      World..  ......
000000F0 00899406 14280000 25090101 00020004     .....(..%.......
00000100 6D61696E 03020100 010E0302 00030408     main............
00000110 00000E05 04000E06 04000407 04080300     ................
```

Figure 11.4 Example output produced by the H option

11.9 Module data – option M

This option displays any header information which is present. This may include version control data, general comments that may have been appended to the file during use of the toolset and copyright information. The data is displayed for individual modules in the object file and includes:

- Module name.
- Transputer type, error mode and occam 2 compiler's 'Y' option - if used indicated by the presence of a 'Y' character.
- Language used.
- Version control data.
- Comments inserted by the toolset, for example, copyright clauses.

Data is displayed in separate blocks for each module. Some of the data is also used by other tools; for example, some comments are used by the debugger tools while version information is used by some tools for compatibility testing.

When *linked* object files are displayed using this option, a long comment will be displayed. This comment gives details of the allocation of memory to each separately compiled code and library module used in the linked module. The following information is given in tabular format:

- Code type - Separately compiled code (SC) or library module (LIB).
- Module name.
- Address offset in linked module.
- Start address.
- End address.
- Reference in library (if applicable) used to locate the relevant library module.

11.9.1 Example module data display

Figure 11.5 shows the module data display for the compiled file `hello.tco`.

```
MODULE:  ANSI_C        ST20 X
VERSION: icc hello.c
COMMENT: INMOS C compiler Version 4.01.09(02:24:52 Mar 11 1995)(SunOS-Sun4)
```

Figure 11.5 Example output produced by the `M` option

11.10 Library index data – option `N`

This option is used to list library indexes. The data is given in a tabular format. For each entry in the index the following information is given:

- Address of the module in the library.
- Symbol name.
- Language.
- Target processor type.
- Error mode.
- occam 2 compiler's '`Y`' option - if used indicated by the presence of a '`Y`' character. The compiler's '`Y`' option causes transputer instructions to be use for channel input/output. If this field is blank this indicates that library calls are being used for channel input/output.

11.10.1 Example library index display

Figure 11.6 shows part of the output produced by the 'N' option for one of the standard C library files.

```
00025C21 ie64op.pax:8340AC71       OCCAM         TA   X
00036155 xlink1.pax:F11BAD5A       OCCAM         TA   X
00034AF7 DATAN2%c                  OCCAM         TA   X
000330D0 DCOS%c                    OCCAM         TA   X
0000B898 DefaultSignalHandler%c    ANSI_C        TA   X
0001CAC6 floorf                    ANSI_C        TA   X
0002007B get_static_size%c         ANSI_C        TA   X
000129AD sub_vfprintf%c            ANSI_C        TA   X
```

Figure 11.6 Example output produced by the N option

11.11 Procedural interface data – option P

This option is only applicable to occam modules or mixed language programs. It displays procedural interface information for all external occam functions and procedures. The following information is displayed for each module:

- Target processor.
- Error mode.
- Whether the Y compiler option was used.
- Language used.
- Amount of workspace used by the procedure or function, in words.
- Amount of vector space used by the procedure or function, in words.
- Parameters used by the procedure or function.
- Data type of parameters.
- Channel usage, if applicable.

Channel usage is displayed in occam notation. A channel marked with an ? is an *input* channel to the code of that entry point, and a channel marked with ! is an *output* channel.

When a library file is listed this will be indicated by the words 'INDEX ENTRY mode:' rather than 'DESCRIPTOR mode'.

11.11.1 Example procedural data display

Figure 11.7 shows an example procedural data display for a compiled occam module. This example is taken from the 'simple' example occam program compiled by oc for the TA processor class.

```
DESCRIPTOR mode: TA    H    language: OCCAM           <ORIGIN DESCRIPTOR>
DESCRIPTOR mode: TA    H    language: OCCAM
ws: 52 vs: 378
PROC simple(CHAN OF SP fs,CHAN OF SP ts)
SEQ
fs?
ts!
:
```

Figure 11.7 Example output produced by the P option

11.12 Specify reference – option R

This option is used in conjunction with any of the other display options to locate a specific symbol within a named library. All library modules that export the symbol are displayed.

The exact format of the display depends on the main display option with which R is used.

Note: Symbol names must be specified in the correct case.

11.13 Full listing – option T

This option displays all *data* found in the input file. Provided that ilist recognizes the file type, the file is decoded in its own format. Text file are displayed as text and unrecognized file types are displayed as a hexadecimal dump.

Data is not displayed in a tabular form but is output in the sequence in which it is found in the file.

The display formats are tailored to each file format and are intended for diagnostic support and analysis; large amounts of data are produced which may require skilled interpretation.

11.13.1 Example full data display

Figure 11.8 shows the full data display for a compiled file hello.tco, showing module information.

```
00000000 LINKABLE
00000002 START_MODULE CORE FMUL FPSUP DUP WSUBDB MOVE2D CRC BITOPS FPTSTERR
 LDDEVID LDMSTVL POP RMC_CORE1 RMC_CORE2 RMC_CORE3 SEMAPHORE DEVICE BIT32 M
S<=28 ICALL X lang: ANSI_C ""
00000010 VERSION tool: icc origin: hello.c
0000001E SYMBOL EXP ROU "main" id: 0
00000028 SYMBOL IMP ROU "_IMS_printf" id: 1
00000039 LOCAL_SYMBOLS number: 2 ids: 2 to 3
0000003C SECTION REA EXE EXP "text%base" id: 4
0000004A SET_LOAD_POINT id: 4
0000004D SYMBOL LOC "local%text" id: 5
0000005B DEFINE_LABEL id: 5
0000005E COMMENT PRINT "INMOS C compiler  Version  4.01.09 (02:24:52 Mar 11
 1995)  (SunOS-Sun4)"
000000AA DEFINE_LABEL id: 0
000000AD LOAD_PREFIX size: 0 SV:2-SV:3 instr: ldc
000000B6 LOAD_TEXT bytes: 2
000000BB DEFINE_LABEL id: 3
000000BE LOAD_TEXT bytes: 1
000000C2 LOAD_PREFIX size: 0 AP(SV:1-LP) instr: call
000000CB LOAD_TEXT bytes: 3
000000D1 ALIGN modulo: 4
000000D4 DEFINE_LABEL id: 2
000000D7 LOAD_TEXT bytes: 16
000000EA COMMENT bytes: 5
000000F4 COMMENT bytes: 37
0000011E END_MODULE
```

Figure 11.8 Example output produced by the T option for a .tco file

11.13.2 Configuration data files

The full data listing of a configured (.cfb) file shows how the processes are mapped onto a transputer system.

11.14 File identification – option W

This option causes the lister to identify the file type. ilist takes a heuristic approach to file identification. The filename is displayed along with the file type. The full path to the file is also displayed if the file is not in the current directory (i.e. if it has been found in the search path specified in the ISEARCH environment variable). This is the default if no other option is supplied.

Table 11.3 indicates how the lister classifies file types.

File format	Default extension	Listed file type
TCOFF compiled unit	.tco	TCOFF LINKABLE UNIT
TCOFF compiled library unit	.lib	TCOFF LINKABLE UNIT LIBRARY
TCOFF linked unit	.lku	TCOFF LINKED UNIT
TCOFF linked library unit	.lib	TCOFF LINKED UNIT LIBRARY
Configuration binary	.cfb	VERSION 1 CONFIGURATION BINARY
Core dump	.dmp	CORE DUMP
Network dump	.dmp	NETWORK DUMP
Extracted SC	.rsc	EXTRACTED SC
Bootable program	.btl	BOOTABLE FILE
ROM bootable program	.btr	BOOTABLE FILE (ROM)
Empty file	–	EMPTY FILE
Text files	–	TEXT FILE
None of the above	–	UNKNOWN BINARY FORMAT

Table 11.3 File types recognized by ilist

where: Version 1 configuration binary files are generated using a T2/T4/T8-series configurer (icconf or occonf).

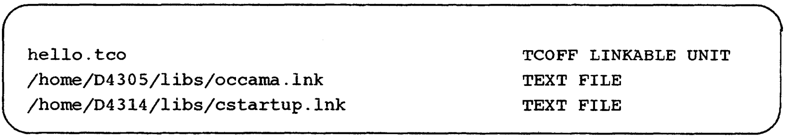
SC files are separately compiled files.

Extracted files are files which have been compiled and developed (using the K option of icollect) for dynamic loading onto a transputer system.

11.14.1 Example file identification display

Figure 11.9 shows the file identification display for the compiled file hello.tco. and two linker control files. This output was generated by the following command:

```
ilist hello.tco occama.lnk cstartup.lnk
```



```
hello.tco                               TCOFF LINKABLE UNIT
/home/D4305/libs/occama.lnk             TEXT FILE
/home/D4314/libs/cstartup.lnk           TEXT FILE
```

Figure 11.9 Example output produced by the W option

11.15 External reference data – option X

This option displays a list of all the code and data symbols imported by the modules specified to the lister, i.e. it lists their external references. External references are references to separately compiled units. For C programs the option will also display any external references to globally visible data.

The output from this option is in a tabular format. It consists of a list of external references and their associated modules. The following information is displayed:

- External reference i.e. name of the separately compiled unit.
- The name of the module in which the external reference exists.
- Language used.
- Target processor.
- Error mode.
- occam 2 compiler's 'Y' option - if used indicated by the presence of a 'Y' character. The compiler's 'Y' option causes transputer instructions to be use for channel input/output. If this field is blank this indicates that library calls are being used for channel input/output.

11.15.1 Example external reference data display

Figure 11.10 shows the external reference data display for the compiled file `hello.tco`.

```
_IMS_printf              <- hello.c                 ANSI_C         ST20 X
```

Figure 11.10 Example output produced by the X option

11.16 Error messages

This section lists error and warning messages that can be generated by the lister. Messages are in the standard toolset format which is explained in appendix A.

11.16.1 Information messages

filename - **Previous message resulted from decoding file as format** *token*

> This message is only output following an error message reported by an internal routine. The information message identifies the file in which the error occurred.

11.16.2 Serious errors

filename - **Cannot close file**

The named file could not be closed.

filename - **Cannot open file**

The named file could not be found/opened for reading/writing.

filename - **Input type invalid with command line options**

The options given to the lister apply to formats incompatible with the file type being read.

Aio failed: *reason*

Failure reported via a low level library routine. An input /output error has occurred.

Aio failed: Bad format: *reason*

Failure reported via a low level library routine. An incorrect file format has been detected - possibly due to file corruption or the use of an incompatible file format.

Cannot close abstract i/o handle *token*
Cannot open abstract i/o handle *token*

An internal structure could not be allocated or released to/from memory.

Parsing command line *token*

The lister was invoked with an unrecognized token on the command line.

11.16.3Fatal errors

filename - **Cannot set file buffer**

Internal error, and error was returned by a call to the C library routine `setvbuf( )` for the file.

Internal errors should be reported to your local SGS-THOMSON distributor or field applications engineer.

12 imakef - makefile generator

This chapter describes the makefile generator imakef that creates makefiles for input to make programs. It explains how the tool can be used to create makefiles and describes the special file naming conventions that allow imakef to create makefiles for mixtures of code types. The chapter describes the format of makefiles generated by imakef and ends with a list of error messages.

12.1 Introduction

make programs automate program building by recompiling only those components whose dependents have been changed since their last build. To do this they read a **makefile** which contains information about the interdependencies of files with one another, along with command lines for rebuilding the program.

imakef creates makefiles for all types of toolset object files, using its built in knowledge of how files referenced within the target file depend on one another. It is intended to be used with all SGS-THOMSON compiler systems that generate TCOFF object code, which includes the the ANSI C compiler icc and the occam 2 compiler oc. Its mode of operation with different languages is controlled by command line options. The makefile is generated in a standard format for input to most make programs.

Makefiles created using imakef are compatible with many public domain and proprietary make programs. The following make programs are directly compatible:

- Borland make.
- UNIX make.
- Microsoft nmake.
- Gnu make.

However, the older Microsoft make program "make" is *not* compatible.

12.2 How imakef works

imakef operates by working back from the target file to determine its dependences on other files, using its knowledge of inputs and outputs of each tool and the compilation architecture of the toolset. For example, compiled object files must be created from language source files using the compiler.

In a similar way linked files must be generated from compiled files. imakef assumes that programs targeted at a single transputer are not required to be configured. Bootable files may therefore be generated from linked units or configuration data files. imakef works back from the target file, determining file dependencies and creating commands to recreate the target file, recompiling and relinking where necessary.

In most cases, imakef is able to discover all the file dependencies required to build a target, in some cases, however, imakef will require some extra pointers. Section 12.4 explains how linker indirect files are used to assist with the building of C programs.

12.3 File extensions for use with imakef

imakef identifies files and file types by a special set of file extensions which identify the transputer target type and compilation error mode. This allows the tool to produce makefiles for mixed module combinations.

Note: The extensions that imakef requires differ in most cases from the standard toolset default extensions which are described in section A.5. For imakef to work correctly the extensions described in section 12.3.1 must be used on all intermediate and target files, at all stages of program development i.e. compiling, linking, configuring, and booting.

The file naming convention uses a three-character extension which identifies the type of file and in most cases includes the transputer target and error mode. Source files for the most part use standard language extensions.

12.3.1 Target files

The following table lists the types of object code files for which imakef can create makefiles, along with the file extension formats that must be used.

Target file	Filename extension
Compiled code.	.t*xx*
Linked code.	.c*xx*
Bootable code for single transputer programs.	.b*xx*
Boot-from-link code for configured programs.	.btl
Boot-from-ROM code, which must be configured.	.btr
Dynamically loadable code.	.r*xx*
Libraries.	.lib
Library usage file.†	.liu
† Creates target file only, does not create makefile.	

Compiled, linked, bootable and non-bootable files, whatever their language origin, have a transputer target designator as the *second* character of the extension, and an error mode designator as the *third* character. Accepted values of these designators are listed below.

2nd Character	Transputer types supported
2	T212, T222, M212
3	T225
4	T414
5	T425, T400
6	T450
8	T800
9	T805, T801
a	Class TA
b	Class TB

3rd Character	Error mode
x	UNIVERSAL
h	HALT
s	STOP

Examples:

.t6x – refers to a compiled module targetted for the T450, in UNIVERSAL error mode.

.t4x – refers to a compiled module targetted for T4 transputers, in UNIVERSAL error mode.

.tah – refers to a module targetted for any 32–bit transputer in HALT error mode.

Compiled code generated by icc is in UNIVERSAL mode, designated by the character 'x'. HALT and STOP code can be generated by the occam 2 compiler oc.

Transputer types are explained further in section B.2.

Program development using imakef and the extensions to use are illustrated in Figure 12.1. Target files which can be created by imakef are shown in bold.

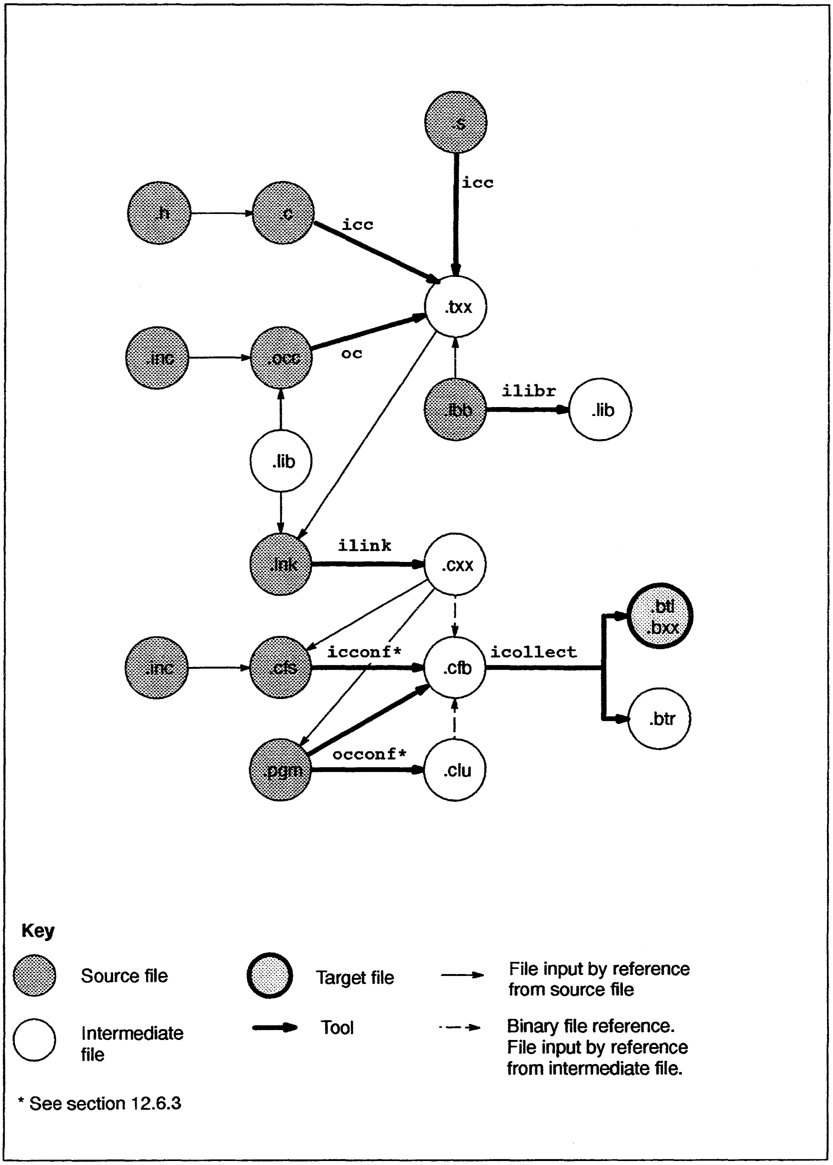


Figure 12.1 Main target files showing extensions required

12.4 Linker indirect files

For C modules linker indirect files must be created for all linked units where imakef will be used to generate a target file. Linker indirect files define to imakef the components of the linked unit, providing a starting point for working out file dependencies.

Linker indirect files must be named after the linked unit to which they relate and carry the .lnk extension.

If the imakef 'C' option is not specified, (see table 12.1) or if imakef cannot find a linker indirect file associated with a particular linked unit, it will assume it is linking occam modules and generate a linker indirect file itself. The file is named after the target filename but is given an extension in the form .l*xx*. The file contains a list of modules to be linked. In addition an #INCLUDE statement references a further linker indirect file, referencing compiler libraries. imakef deduces the compiler libraries to be included from the extension of the linked object file.

Section 12.6.2 provides a short description of linker indirect files and several examples are given in section 12.7.

12.5 Library indirect and library usage files

When building a library using imakef, a file must be provided that contains the names of all the object modules required to build the library. This file is known as a library indirect file and has the extension .lbb. See chapter 9 for further details.

Library usage files describe the dependencies of a library on other libraries or separately compiled code. They contain a list of files to which the library must be linked before it can be run, and ensure that the correct linker commands are generated.

Library usage files should be created for all user-defined libraries where the source of the library is not available. They are created using imakef.

Library usage files are given the same name as the library to which they relate, but with a .liu extension. To create a library usage file using imakef, specify the library name and add a .liu extension. For example, the following command creates a library usage file for the library mylib.lib:

```
imakef mylib.liu
```

When imakef is used to create a library usage file no makefile is generated.

12.6 Running the makefile generator

The imakef tool takes as input a list of files generated by tools in the toolset and generates a makefile, containing full instructions of how to build the application program. The output file is named after the first target filename and is given a .mak extension (if no output file is specified on the command line).

12.6 Running the makefile generator

To invoke `imakef` use the following command line:

▶ `imakef` *filenames* { *options* }

where: *filenames* is a list of target files for which makefiles are to be generated. If more than one file is specified the single makefile generated will generate all of the specified files.
options is a list, in any order, of one or more options from Table 12.1.

> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be given in any order.
>
> Options may be entered in upper or lower case and can be given in any order.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
C	Specifies that the list of files to be linked is to be read from a linker indirect file. This option must be specified when the program includes C modules.
D	Disables all debugging support in a makefile. The default is to build the makefile with full debugging information to support post-mortem debugging.
GA	Enable the use of the INQUEST debugger in interactive mode. This option is incompatible with the 'Y' and 'NV' options.
I	Displays full progress information as the tool runs.
M	Produce compiler, linker and collector map files for `imap`.
NC	Indicates that an NDL file is referenced and therefore either the `inconf` or `onconf` configurer will be used. Not applicable to T2/T4/T8-series transputers.
NI	Files in the directories in `ISEARCH` are not put into the makefile. This means that system files are not present, making it easier to read.
NIF	Generate a Network Initialization file. The name is derived from the name of the NDL file, referenced within the configuration description. This option is only valid if specified in conjunction with 'NC'. Not applicable to T2/T4/T8-series transputers.
NV	Generates a configuration which does not use software virtual routing. Only valid when used with `icconf` or `occonf`.
O *filename*	Specifies an output file. If no file is specified the output file is named after the target file and given the `.mak` extension.
R	Writes a deletion rule into the makefile.
Y	Applies only to occam modules. Disables the use of library calls for channel input/output and instead uses transputer instructions. This option is incompatible with the 'GA' option.

Table 12.1 `imakef` options

12.6.1 Example of use

```
imakef hello.b6x –c
```

This creates the makefile `hello.mak` which when used as input to make generates the bootable file `hello.b6x` (a bootable file for T450 transputers).

12.6.2 Specifying language mode

imakef can be used with all compilers in the SGS-THOMSON TCOFF family. This includes the ANSI C compiler icc and the occam compiler oc.

imakef has two modes of operation: one for the traditional languages such as C, and another for occam. occam mode is the default; C operation is controlled by a command line option.

In occam programs, file dependencies are wholly deducible from the source and target files. In C programs the list of files to be processed by the linker must be given in a linker indirect file; use of the imakef 'C' option instructs imakef that there are linker indirect file(s) to be read. The linker indirect files must include all the components of a program, including any libraries that are used.

The 'C' option must be specified for all C programs and for any mixed language programs which incorporate modules in these languages. For mixed language programs all files which are to be linked must be listed in the linker indirect file(s), including any occam modules or library files. In systems that use mixtures of code compiled for different transputer types and error modes, a separate linker indirect file must be created for each.

The C option has two effects:

- It directs imakef to search for a linker indirect file with an extension of .lnk. If no .lnk file is found then imakef looks for an occam source file. The default is to only search for an occam source file.
- It directs imakef to search for a .cfs configuration file before looking for a .pgm configuration file. The default is to search in the opposite order. The filename extension of the configuration source file found determines which configurer imakef will apply. If a .pgm file is found then an occam configurer will be used (occonf), while if a .cfs file is found then a C configurer will be used (icconf).

An example is given in section 12.7.3 of how imakef may be used to build a mixed language program.

12.6.3 Configuration description files

When imakef builds a makefile for a configured program it will look for the presence of a configuration description file which has the same filename root as the program to be built, but a different filename extension.

The type of file searched for depends on the mode of operation specified to imakef. If the default occam mode is used, that is, the 'C' option is not specified, imakef will look first for a configuration description file with the extension .pgm, (readable by the occam configurer occonf). If a .pgm file is not present imakef will then look for a .cfs file (readable by the 'C-style' configurer icconf).

If the C mode is used, that is, the 'C' option is specified, the reverse sequence is used, that is, imakef looks first for a .cfs file.

Which configurer is targeted is also dependant on the presence, or not, of the 'NC' option.

12.6.4 Debug data

By default the INQUEST debugger is only supported in post-mortem mode. Interactive debugging has to be specifically enabled using the 'GA' command line option.

The 'D' option disables the generation of all debugging support in the target file. If this option is used the resulting target code cannot be debugged.

12.6.5 Software virtual routing and channel input/output

The option 'NV' disables the use of software virtual routing processes in target configuration files, configured by `icconf` or `occonf`.

If the 'NV' option is specified for non-configured targets, it is ignored.

The 'Y' option instructs `imakef` to use transputer instructions for channel input/output rather than using library calls.

The 'Y' option only applies to occam modules and if specified the occam compiler and configurer and the linker will be invoked with the 'Y' option. If either `icconf` or `occonf` are used then `imakef` will automatically invoke them with the 'NV' option, even if it is not specified on the command line. This maintains consistency in the implementation of channel input/output because the software virtual routing processes require channel input/output to be implemented by library calls. If the 'Y' option is used it will not be possible to use the software virtual routing processes and they may therefore be disabled.

12.6.6 Boot-from-ROM target files

If a specified target file has a 'BTR' extension, a makefile will be produced for a boot-from-ROM target file. `imakef` will insert a macro '`$ (CONFOPTROM)`' to invoke the configurer with the 'RA' option. All toolset configurers currently use the 'RA' option for applications which will be executed in RAM.

The user may wish to edit the macro entry in the makefile, in order to specify different configurer command line options which control boot-from-ROM output. This will depend upon the configurer used. See section 12.8.1 for further details of macros.

12.6.7 Removing intermediate files

Intermediate files can be removed by specifying the 'R' option to `imakef`. This adds a *delete* rule to the makefile which directs `make` to remove all intermediate files after the program is built. The delete operation is only honored if `make` is subsequently invoked with the DELETE target.

12.6.8 Files found on ISEARCH

When `imakef` runs, it includes all dependencies in the set of rules. The `NI` option prevents `imakef` recording in the makefile, any dependencies on files found using `ISEARCH`. As a result the makefile is easier to read and is more portable.

12.6.9 Map file output for imap

Using the 'M' option, imakef can be made to generate switches in the calls to the compiler, linker or collector to output map files.

These map files are then available for reading by the imap tool, details of which can be found in chapter 13.

12.7 imakef examples

This section contains several examples which use imakef with different SGS-THOMSON toolsets. The examples aim to demonstrate imakef and may not be specific to your toolset.

12.7.1 C examples

The first example shows how to create a makefile for a multi-module program, written in C, running on a single T450 target. The second example shows how to create a makefile for a configured C program.

Single transputer program

This first example is for a program which is not configured.

The example program is made up of three source files, written in C:

```
main.c
hellof.c
worldf.c
```

imakef needs to know the names of the main components of the program, and looks for the associated linker indirect file hello.lnk:

hello.lnk must contain the following text:

```
main.t6x
hellof.t6x
worldf.t6x
#include cnonconf.lnk
```

Note: the use of the .t6x extension rather than .tco. This is because imakef needs to work out the required processor type. The C run-time start-up linker indirect file cnonconf.lnk is also included. The inclusion of this file is standard for all C programs which are not configured and directs imakef to include the libraries. To create the makefile use the command:

```
imakef hello.b6x -c
```

Note: the use of the .b6x extension instead of .btl. Using this form of extension informs imakef that we wish to create a bootable program for a single T450 transputer without the aid of the configurer. The makefile hello.mak is created.

Multitransputer program

This example program uses the configurer to place linked units on two processors. The program is made up of the following source files written in C:

```
master.c
mult.c
multi.cfs
```

The `.cfs` file is the configuration description file. It places two linked units on two processors, using the following statements:

```
use "master.c8x" for master;
use "mult.c4x" for mult;
```

`Note:` the use of the `.cxx` form of extension instead of the toolset default extension for linked units `.lku`. `imakef` reads the `.cfs` file and determines that the program is made up of two linked units, each of which must have an associated linker indirect file, namely, `master.lnk`, and `mult.lnk`.

The two linker indirect files files must contain the following text:

master.lnk:

```
master.t8x
#include cstartup.lnk
```

mult.lnk:

```
mult.t4x
#include cstartrd.lnk
```

Again note the use of the `.txx` form of extension. `master.lnk` includes the C run-time start-up linker indirect file `cstartup.lnk`, which is used for configured programs linked with the full ANSI C run-time library. `mult.lnk` includes `cstartrd.lnk`, the standard C run-time start-up linker indirect file used for configured programs linked with the reduced library. This library can be used by `mult.t4x` because the module does not require host access.

To create the makefile use the following command:

```
imakef multi.btl –c
```

The `.btl` extension informs `imakef` that the target is a configured program, to be built from a configuration description file called `multi.cfs`. The makefile `multi.mak` is created.

12.7.2 occam examples

Two examples are described, the first for a multi-module program running on a single transputer and the second example for a configured program.

Single transputer program

The example program is made up of four source files, written in occam:

```
sorthdr.inc
element.occ
inout.occ
sorter.occ
```

To create the makefile use the following command:

```
imakef sorter.b5h
```

Note the use of the .b5h extension instead of .btl. Using this form of extension informs imakef that we wish to create a bootable program for a single transputer without the aid of the configurer.

The makefile generator has built-in knowledge of the file name rules for occam. In this example, it knows by examining the file name that the program to be built is for a single T400 or T425 processor in HALT mode, and that the source of the main body of the program is in the file sorter.occ. It reads the file sorter.occ and discovers that it uses a library called hostio.lib, the two compilation units inout.tco and element.tco, and two include files, sorthdr.inc and hostio.inc. It then reads the sources of the include files and compilation units and finds no more file dependencies.

With this information about source file and their dependencies, imakef builds a makefile called sorter.mak containing full instructions on how to build the program and creates a linker indirect file sorter.l4h (see section 12.4).

To build the program run the make program on sorter.mak. The entire program will be automatically compiled, linked and made bootable, ready for loading onto either a T400 or T425 transputer.

Multitransputer program

This version of the sorter program is configured to place linked units on four processors.

The program is made up of the following occam files:

```
sorthdr.inc
element.occ
inout.occ
sortmak.pgm
sortsoft.inc
```

To create the makefile use the following command:

```
imakef sortmak.btl
```

The .btl extension informs imakef that the target is a configured program, to be built from a configuration description file called sortmak.pgm. The configuration description references two linked units:

```
#USE "inout.cah"
#USE "element.cah"
```

Note: the use of the .cxx form of extension instead of the toolset default extension for linked units .lku. imakef reads the .pgm file and will produce a file called sortmak.mak suitable for building the program.

To build the program run the make program on sortmak.mak.

12.7.3 Mixed language program

This example, uses a mixed language program which combines both occam and C modules.

The example program is made up of the following files:

mixed.t6h	– Compiled occam module
cfunc.t6x	– Compiled C module

where: the occam module is the main program which calls in the C function.

To create the makefile for the example program use one of the following command:

```
imakef mixed.b6h -c
```

This command informs imakef that we wish to create a bootable program for a single T450 processor in HALT mode. The 'c' option tells imakef to search for a linker indirect file.

imakef needs to know the names of the C components of the program, and looks for the associated linker indirect file mixed.lnk. Because a linker indirect file is supplied to imakef, all the modules to be linked must be listed.

mixed.lnk must contain the following files:

```
mixed.t6h
cfunc.t6x
callc.lib
hostio.lib
#INCLUDE clibsrd.lnk
#INCLUDE occam450.lnk
```

The occam module is listed first, because it contains the main entry point of the program. **Note:** the use of the .t6h and .t6x extensions. The C module will be compiled in UNIVERSAL mode, which is the standard mode for the C compiler. This does not cause a problem because UNIVERSAL mode may be called by HALT mode.

The files hostio.lib and callc.lib are the occam libraries. occam450.lnk contains a list of occam compiler libraries which may be required.

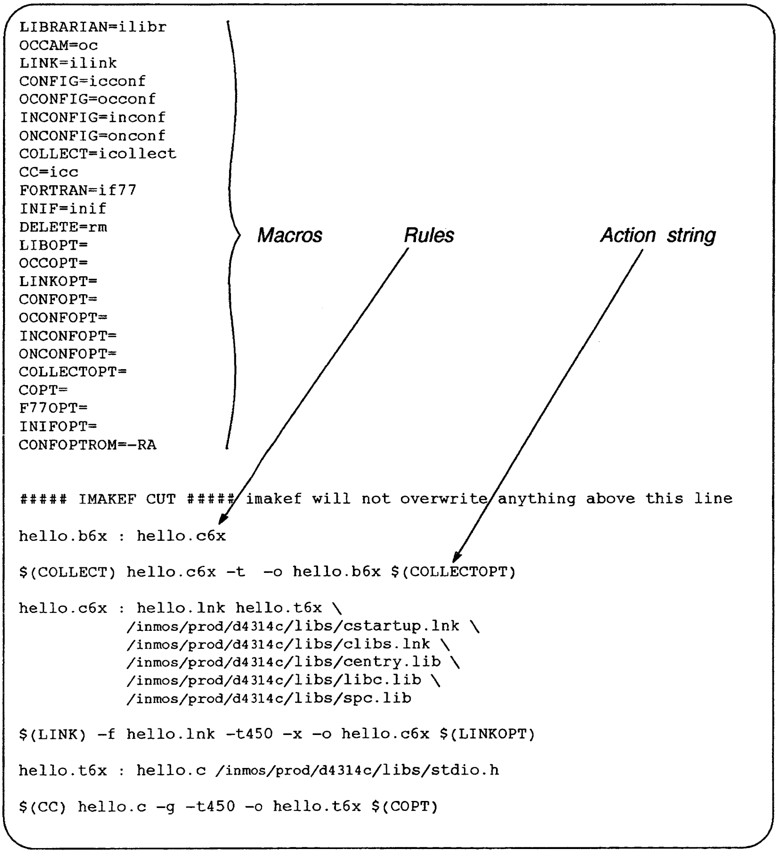
clibsrd.lnk references the reduced C runtime library used by the C module.

With this information **imakef** builds the makefile **mixed.mak**.

Further information about mixed language programming can be found in the accompanying '*User Guide*', where it is supported by the toolset.

12.8 Format of makefiles

Makefiles essentially consist of a number of *rules* for building all the parts of a program. Each rule contains two main elements: a definition of the file's dependencies in a format acceptable to `make` programs; and the command to recreate the file on a specific host. All makefiles also contain macros which define command strings and option combinations.



```
LIBRARIAN=ilibr
OCCAM=oc
LINK=ilink
CONFIG=icconf
OCONFIG=occonf
INCONFIG=inconf
ONCONFIG=onconf
COLLECT=icollect
CC=icc
FORTRAN=if77
INIF=inif
DELETE=rm
LIBOPT=
OCCOPT=
LINKOPT=
CONFOPT=
OCONFOPT=
INCONFOPT=
ONCONFOPT=
COLLECTOPT=
COPT=
F77OPT=
INIFOPT=
CONFOPTROM=-RA

##### IMAKEF CUT ##### imakef will not overwrite anything above this line

hello.b6x : hello.c6x

$(COLLECT) hello.c6x -t  -o hello.b6x $(COLLECTOPT)

hello.c6x : hello.lnk hello.t6x \
           /inmos/prod/d4314c/libs/cstartup.lnk \
           /inmos/prod/d4314c/libs/clibs.lnk \
           /inmos/prod/d4314c/libs/centry.lib \
           /inmos/prod/d4314c/libs/libc.lib \
           /inmos/prod/d4314c/libs/spc.lib

$(LINK) -f hello.lnk -t450 -x -o hello.c6x $(LINKOPT)

hello.t6x : hello.c /inmos/prod/d4314c/libs/stdio.h

$(CC) hello.c -g -t450 -o hello.t6x $(COPT)
```

Figure 12.2 Example makefile

12.8.1 Macros

All makefiles created by `imakef` include a set of macro definitions inserted at the head of the file.

Macros define strings which are used to call the compiler, the configurer, the linker, the librarian, the collector, and the eprom formatter tools, and fixed combinations of options for these tools.

Macros are provided so that customized versions of the toolset commands, and specific combinations of options, can be easily incorporated. Existing macros can be modified for specific host environments, and new macros created, by editing the makefile.

The full set of macros defined by `imakef` can be found by consulting any makefile created by the tool.

12.8.2 Rules

Rules define the dependencies of object files on other files and specify *action strings* to build those files.

Example:

```
example.t6h : example.c
    $(CC) example -t6 -h -o example.t6h $(COPT)
```

This rule first defines the target as the compiled program `example.t6h`, which is dependent on the source file `example.c` and then specifies the command that must be invoked to build it.

The first rule in all makefiles is for the main target. Succeeding rules define sub-components of the main target, and are listed hierarchically.

Action strings

Action strings define the complete command line needed to recreate a specific file. The format is similar for all tools and consists of a call to the tool via a predefined macro, a fixed set of parameters, a list of command line options, usually via a macro, and the output filename. (The output file is specified on the command line so that the rebuilt file is always written to the directory that contains the source.)

12.8.3 Delete rule

The delete rule directs `make` to remove all intermediate object files once the program has been built. It consists of a single labelled action string which invokes the host system 'delete file' command. Deletion is only performed if `make` is subsequently invoked with the DELETE target.

The delete rule is appended to the makefile by specifying the `imakef` '`R`' option.

12.8.4 Editing the makefile

Makefiles created by the `imakef` tool can be edited for specific requirements. For example, new macros can be added and new rules defined for compiling and linking code written in other languages.

Adding options

imakef generates action strings which have the minimum of options for each tool. In most cases additional options are unnecessary or may be specified using compiler directives. For example, the D and U options to icc may be replaced by #define and #undef at the start of the C source. Several options to oc may be replaced by #OPTION statements at the start of occam source.

To modify the set of default options for a particular tool simply edit the appropriate macro in the makefile.

For example, if the output of progress information is to be enabled for all invocations of the compiler, the compiler 'I' option would be added to the macro which defines the standard combination of options for invoking the compiler. So for the example makefile in figure 12.2 the compiler macro 'CC = icc' could be redefined to be 'CC = icc –i'. Alternatively a new macro containing only the 'I' option could be defined and added to each compiler action string.

Re–running imakef

Once the set of options have been changed in the macros, it is useful to retain this set of options when imakef is run again. For this reason, imakef will check for the existence of a previous makefile. If one exists, it will re-use (in the new makefile) the set of macro definitions from the old one, plus any additional text up to the line:

```
##### IMAKEF CUT ##### imakef will not overwrite anything above this line
```

12.9 Error messages

imakef generates error messages of severities *Warning* and *Error.* Messages are displayed in standard toolset format.

12.9.1 Warnings

#IMPORT references are illegal in configuration text

At the given line number in the file there is a reference to the #IMPORT directive, which is illegal for configuration source.

#INCLUDE may not reference a library

The #INCLUDE directive is being used to reference a file with the .lib extension.

#INCLUDE may not reference binary files

The #INCLUDE directive is being used to reference a file containing compiled code.

#USE may not reference source files

Applies to occam modules only.
The directive #USE cannot be used to reference occam source code.

Cannot write library usage file

The library usage file cannot be opted for writing.

Cannot write linker command file

The linker command file cannot be opened for writing by the program.

Empty library usage file created

An empty library usage file has been created because there is no `.lbb` file or there are no library dependencies that make up the library.

Error whilst reading

A file system error has occurred whilst reading the source.

File generated even though NI option supplied

An option, e.g. '`NIF`' has been specified in order to generate a file but the file is located via `ISEARCH`. The '`NI`' option is ignored with respect to this file.

Found NETWORK directive but not using NDL configurer, need imakef NC option

A `#NETWORK` (C) or `#network` (occam) directive in the configuration description implies that the target processor is an IMS T9000 transputer. Therefore the `NC` option should be used.

GA option has no effect with the Y option

The `GA` and `Y` options are incompatible.

Incomplete compiler directive

At the given line number in the file there is an invalid compiler directive.

Invalid syntax in environment variable

`ISEARCH` has been set up incorrectly.

Invalid to mix STOP and HALT error mode

The STOP and HALT error modes are mutually exclusive. Make sure the `#USE` statements in the configuration description are compatible.

Invalid to mix T9000 linked units with any other processor types

A T2/T4/T8-series transputer network can talk to an IMS T9000 transputer network via an IMS C100 system protocol converter but they cannot be mixed within the same network at configuration level.

NV option has no effect with the NC option

The `NC` option causes either of the configurers `inconf` or `onconf` to be selected, neither of which have a `NV` command line option.

Source file does not exist

The referenced source file does not exist.

Unexpected file type in INCLUDE directive

INCLUDE directives are supported by the C and occam compilers and configurers. They are used to import source code e.g. definitions of constants and standard header files. Include files usually have a `.inc` extension.

Unexpected/unknown file type in USE directive

The occam compiler's USE directive should only be used to reference compiled source files `.txx` or library files `.lib`. Check the extension follows the naming rules for `imakef` extensions.

12.9.2 Errors

A library usage file is only valid by itself

If a library usage file is specified as the target file, no other target file may be specified.

Cannot have a makefile

The file specified on the command line is not one for which `imakef` can generate a makefile. `imakef` can only create makefiles for object files, bootable files and library usage files.

Cannot open "*filename*" :*reason*

The file specified as the output file cannot be opened for writing by the program, for the reason given.

Command line is invalid

An incorrect command line was supplied to the program. Check the syntax of the command and try again.

Creating root node of file structure
Malloc failed
Out of memory

The program has failed while trying to dynamically allocate memory for its own use. Try using a transputer board with more memory. If the program is being run on the host it may be possible to increase the memory available using host commands.

Target is not a derivable file

The specified file cannot be generated by the toolset.

Tree checking failed - no output performed

The tree of files has been found to be invalid and unusable for generating makefile. This message always follows a message indicating what is wrong with the tree. The most common reason for this error is the presence of cyclic references in the source.

"*filename*" **unknown/illegal file reference**

A compiler directive is attempting to reference the wrong type of file.

Writing file

A host system error occurred while the file was being written.

13 imap - memory mapper

This chapter describes the memory map tool **imap**. The tool takes the text output from the toolset compiler, linker and collector and gives the absolute addresses of the static variables and functions. The chapter begins with an introduction to **imap** and explains the command line syntax. **imap**'s output is described in some detail and an example is given. The chapter ends with a list of error messages.

13.1 Introduction

The **imap** tool takes as input memory map files output by the compiler, linker and collector. Command line options for these tools enable the user to specify that a memory map file is to be produced. **imap** collates the information from the different source files and puts it in a format suitable for output on the display screen. Alternatively the output from **imap** can be redirected to another output file as the user wishes. Memory maps may be generated for both single and multiprocessor transputer programs.

imap is invoked by supplying it with the name of a map file produced by the collector. The tool will automatically determine the names of map files produced by the linker and compiler, provided the naming convention for extensions has been adhered to, see section 13.2.1. Each tool generates a different level of information:

Collector: For each process on each processor, memory locations of code, static, heap, stack, invocation stack and vector space are listed.

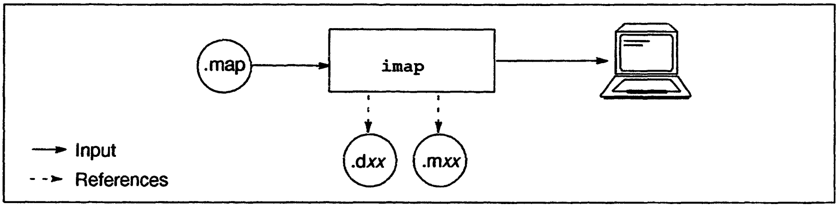
Linker: For each process the offset in memory for code and static for individual modules which make up the process are listed.

Compiler: For each module listed in the linker output file the offset in memory for individual static items, procedures and functions are listed.

Where a particular category of information is not applicable to a language, this field will be left blank. occam programs for instance do not use heap, so obviously such details are not generated for occam.

Where the output files from the compiler and linker cannot be opened or parsed properly **imap** will insert a warning at the appropriate point in the output. Specific addresses of static data or functions associated with that file will not be given.

The operation of the map tool in terms of standard toolset file extensions is shown below. Output is sent to standard out, which is usually set to the display screen.



13.2 Running the map tool

To invoke the map tool use the following command line:

► `imap` *filename* { *options* }

where: *filename* is the name of the file containing the map output from the collector `icollect`. If there is no extension given, `.map` is assumed. Otherwise the file name is taken as given.

options is a list of the options given in Table 13.1.

> Options must be preceded by '–' for UNIX-based toolsets and either '–' or '/' for MS-DOS based toolsets. **Note:** '–' is used in all documentation examples.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.
>
> Options may be supplied in an indirect argument file, prefixed by '@', (see section A.1.2 for details)

Only one filename may be given on the command line.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
`A`	Displays the list of symbols produced by the linker, including those symbols the linker identifies as not being used. This option will not override the '`R`' option if it is used.
`I`	Displays progress information as `imap` processes information from the input files, such as the filenames of files as they are opened and closed.
`O` *filename*	Specifies an output file.
`P` *processor name*	Limits the output generated by `imap` to that associated with the specified processor. *processor name* is defined by the user in the configuration description.
`R`	This option reduces the amount of detail generated by `imap` in two ways: • the Module memory usage table only displays details for user processes. • the Symbol table excludes those symbols containing a '%' character in their name. Such symbols are normally internal symbols e.g. C runtime library symbols.
`ROM` *hex offset*	This option is only applicable to, and must be specified for, code targetted at ROM. It enables a hexadecimal offset to be specified which represents the start address of the code in ROM. This offset will be added to the start address of any code which is to run in ROM, in `imap`'s output.
`SN`	Sorts the symbol tables by symbol name in ASCII order.

Table 13.1 `imap` command line options

Examples of use:

```
imap myprog

imap myprog.map
```

Both the above examples will cause imap to read the file myprog.map, generated by the collector.

13.2.1 Source files required by imap

Three different types of source file are read by imap and should therefore be made available. The files are in fact memory maps generated by the compiler, linker and collector. The appropriate command line option must be specified on each tool's command line including a filename for the map produced. The filename specified by the user must have the appropriate extension as indicated in table 13.2.

Extension	File description
.mxx	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. For example if the compiler object file takes the default extension .tco, the information file is given the extension .mco.
.dxx	Map file output by the linker. The characters '*xx*' are determined by the 2nd and 3rd characters of the extension given to the linker output file. For example if the linker output file takes the default extension .lku, the information file is given the extension .dku.
.map	Map file output by the collector.

Table 13.2 Files extensions for imap source files

13.2.2 Re-directing imap's output

imap's output goes to standard out by default. To redirect it to an output file, use the 'o' option and specify an output filename.

13.3 Output format

This section describes the format of the memory map produced by imap. An example output is given in section 13.4.

If the imap tool cannot find a linker or compiler output file, it will insert a warning message in place of the missing information. It will not produce a warning if the process or module comes from a library (such as the system process library).

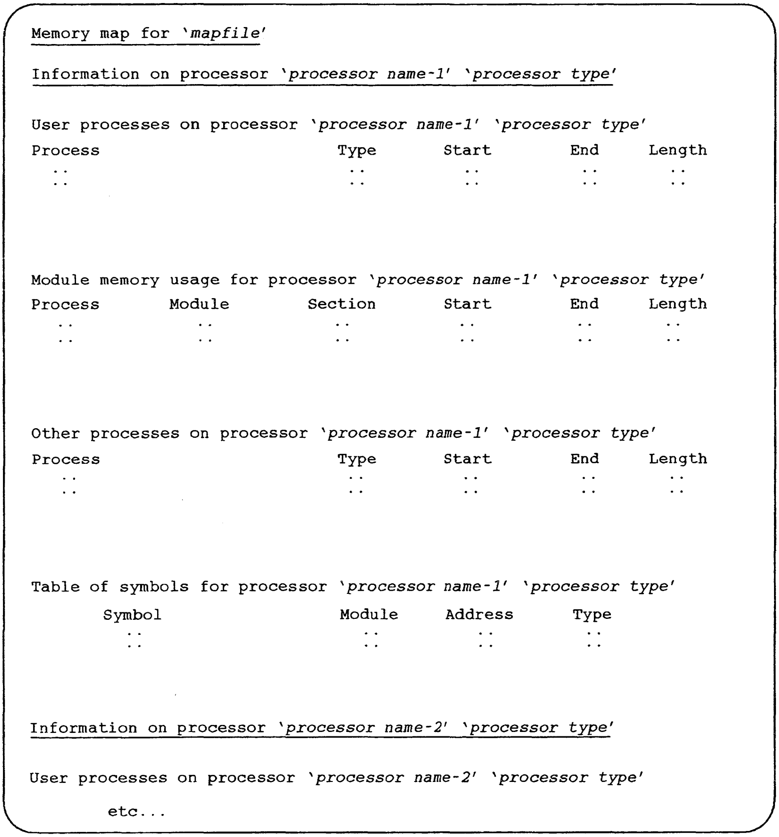
13.3.1 imap memory map structure

Information is given for each processor in turn and includes the following:

- a table of the memory blocks that user processes use;
- a table of the code and static memory blocks that each module uses;
- a table of the memory blocks that non-user processes use;
- a table of symbols used by the processor.

Note: the '`P`' command line option can be used to limit **`imap`**'s output to a specified processor.

Figure 13.1 shows the layout of **`imap`**'s output.



```
Memory map for 'mapfile'

Information on processor 'processor name-1' 'processor type'


User processes on processor 'processor name-1' 'processor type'
Process                        Type       Start        End       Length
  ..                            ..         ..          ..         ..
  ..                            ..         ..          ..         ..




Module memory usage for processor 'processor name-1' 'processor type'
Process       Module        Section       Start        End       Length
  ..            ..            ..           ..          ..         ..
  ..            ..            ..           ..          ..         ..




Other processes on processor 'processor name-1' 'processor type'
Process                        Type       Start        End       Length
  ..                            ..         ..          ..         ..
  ..                            ..         ..          ..         ..




Table of symbols for processor 'processor name-1' 'processor type'
       Symbol                  Module     Address      Type
         ..                      ..         ..          ..
         ..                      ..         ..          ..



Information on processor 'processor name-2' 'processor type'

User processes on processor 'processor name-2' 'processor type'

       etc...
```

Figure 13.1 **`imap`** output format

The name of the map file input on **`imap`**'s command line, is given at the top of the memory map file.

Each table identifies the processor it relates to and gives the processor's type e.g. ST20, T450, T805 etc. The name of the processor is taken from the name specified in the configuration description file.

All tables, except the symbol table, are sorted on start address order. Each of the tables is described below.

13.3.2 User processes

The table headed with "User processes" gives the start and end addresses and lengths of the various blocks of memory used by the user processes for that processor. The table is ordered by start address and is structured as follows:

- Process name or 'All' if it is the parameter data block, which is not associated to just one process. 'M:' indicates that the block's process has no name and the list that follows gives the names of the linked units that made up the process.
- Memory block type – `code`; `heap`; `overhd`; `param`; `stack`; `static` or `vector`. See table 13.3.
- Start address in hexadecimal.
- End address in hexadecimal.
- Length in decimal.

Block type	Description
`code`	Used for code
`heap`	Used for heap
`overhd`	Used for invocation stack
`param`	Used for the parameter data block
`stack`	Used for workspace
`static`	Used for static data
`vector`	Used for vector space (for occam programs)

Table 13.3 Memory block types

13.3.3 Module memory usage

The table headed with "Module memory usage" gives the memory areas that are used by each module for code or static data. The table is ordered by start address and has the following format:

- Process name.
- Module name. Modules which come from a library are prefixed by 'L:'.
- Section name to which the area belongs.

- Start address in hexadecimal.
- End address in hexadecimal.
- Length of the area in decimal.

Examples of section names are `pri%text%base` and `text%base` for code, and `static%base` for static data.

If the 'R' command line option is used only details of user processes are shown.

13.3.4 Other processes

The table headed with "Other processes" is the same as the "User processes" table but for all the non-user processes. These are system processes added by the configurer. This table will not include an entry for the parameter data block.

13.3.5 Symbol table

The symbol table, by default is sorted in address order. The user can specify it is to be sorted by symbol name into ASCII order, by specifying the 'SN' command line option. The symbol table gives the following information:

- Symbol name.
- Module name.
- Start address associated with symbol (in hexadecimal).
- Symbol type (see below).

The type field of the symbol table is either taken directly from the compiler map file, or is created by **imap**. In the latter case, the field will be enclosed in parentheses. This information is based on which section the symbol comes from. Refer to the compiler documentation for the meaning of items in this field that aren't enclosed in parentheses.

Note: command line options can be used to extend or limit the amount of symbol information generated. Normally **imap** only gives details of symbols used by the program; the 'A' option instructs **imap** to include unused symbols in the list. The 'R' option prevents details of internal symbols, such as those used by the runtime libraries, being listed.

13.4 Example

The following example output is only a partial listing, all the table entries have been truncated in order to demonstrate the overall layout of a map file. The example is based on the 'hello.c' program configured for a single ST20 processor. The example was generated by the command:

```
imap hello
```

```
Memory map for 'hello'
======================

Information on processor 'Single' (ST20):
=========================================

User processes on processor 'Single' (ST20):
--------------------------------------------
                  Process                    Type   Start      End      Length
---------------------------------------- ------ --------- --------- --------
Simple                                   stack  #80000170 #80001183     4116
Simple                                   overhd #80001184 #80001197       20
Simple                                   code   #80001198 #80003E37    11424
Simple                                   static #80003E38 #80004197      864
Simple                                   heap   #80004198 #80010997    51200
All                                      param  #80010998 #80010AC3      300

Module memory usage for processor 'Single' (ST20):
--------------------------------------------------
     Process          Module        Section        Start      End      Length
------------------ ------------ -------------- --------- --------- --------
Simple             hello.c      text%base      #00001198 #000011B3       28
Simple             L:centryd1.p text%base      #000011B4 #000011C7       20
Simple             L:centryd2.p text%base      #000011C8 #0000126B      164
Simple             L:ioprgnam.c text%base      #0000126C #000014D3      616
 --                   --          --              --        --          --

Simple             L:ioprgnam.c static%base    #00003E38 #00003E3F        8
Simple             L:centryd2.p debug%1        #00003E38 #00003E3B        4
Simple             L:ioprgnam.c debug%1        #00003E3C #00003E4F       20
 --                   --          --              --        --          --

Simple             L:ioinit.c   static%base    #00004108 #00004117       16
Simple             L:malloc.c   static%base    #00004118 #00004123       12
Simple             L:exit.c     static%base    #00004124 #00004127        4
Simple             L:fatal.c    static%base    #00004128 #00004197      112

Other processes on processor 'Single' (ST20):
---------------------------------------------
                  Process                    Type   Start      End      Length
---------------------------------------- ------ --------- --------- --------
Init.system.simple                       stack  #80000170 #80000193       36
Init.system.simple                       overhd #80000194 #800001B7       36
Init.system.simple                       code   #800001B8 #8000023F      136
System.process.b                         stack  #80000240 #8000025F       32
System.process.b                         overhd #80000260 #80000273       20
System.process.b                         code   #80000274 #800002E3      112

Table of symbols for processor 'Single' (ST20):
-----------------------------------------------
                  Symbol                     Module     Address     Type
---------------------------------------- ------------ --------- -----------
CENTRYD_stage2%c                         libc.lib     #80001198 (code)
ChanIn%c                                 libc.lib     #80001198 (code)
ChanOut%c                                libc.lib     #80001198 (code)
 --                   --          --              --        --          --

main                                     hello.c      #80001198 code
exit                                     spc.lib      #8000119A (code)
vlinkfunc_out%                           libc.lib     #8000119A (code)
_IMS_fdmax                               libc.lib     #8000410C (static)
_IMS_closeptr                            libc.lib     #80004114 (static)
```

Figure 13.2 Example partial listing of imap's output

13.5 Error messages

This section lists each error message that can be generated by the memory map tool. Messages are in the standard toolset format which is explained in appendix A.

All open files are closed when an error is found and the tool halts without producing a map.

13.5.1 Serious errors

Filename **input file cannot be parsed properly**

The named file cannot be read by `imap`.

Cannot open collector's output file for reading

The collector map file specified on the command line cannot be found. Check that the extension used for the collector map file is in the correct format. See section 13.2.

Cannot open output file for writing

The output file cannot be opened for writing. May indicate a disk space problem or some other host system error.

Error parsing command line

The command line has the wrong syntax or a non–existent option has been specified.

Must specify input file

An input file must be specified.

Only single output filename allowed

More than one output filename has been specified.

Only single processor name allowed

More than one processor name has been specified with the '`P`' option; this is illegal.

Only single ROM offset value allowed

More than one ROM offset has been specified.

13.5.2 Fatal errors

Filename **internal data structure failure or file corrupt**

A source file used by `imap` has referenced something which cannot be found. This can occur when redundant map files are read by `imap` in error.

Filename **out of heap space**

There is not enough heap space to generate the memory map.

Unexpected end of file

A source file, read by `imap` has been corrupted. Regenerate compiler, linker and collector map files.

14 imem450 - memory interface configurer

This chapter describes 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*. It can create and modify the file interactively, displaying the effects that the parameters would produce.

The contents of the memfile are used to create the data to initialize the memory interface from a host link or from a ROM. The tool can also be used, interactively or in batch mode, to produce timing data or waveform diagram files for printing.

This chapter describes imem450 and outlines its capabilities. Example displays are provided. The chapter ends with a list of error messages.

The format of a memfile is described and an example is given in Appendix E. The process of converting a memory design to a memory interface configuration is described in the '*ST20 Toolset User Guide*'.

14.1 Introduction

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 to be defined with different interface and timing characteristics. The memory interface has a 32-bit data bus and each bank of memory can be configured to be 32-bit, 16-bit or 8-bit wide.

The memory interface of the ST20 can be initialized by writing to memory-mapped configuration registers. This is performed by the bootstrap code initializing the ST20, which may be loaded from a remote host or read from a ROM.

The imem450 tool can be used to create and modify a memfile and display or print the timing data and waveforms it would produce.

The operation of imem450 in terms of input and output files is shown in figure 14.1. The files are shown with their standard file name extensions, which are recommended but not required.

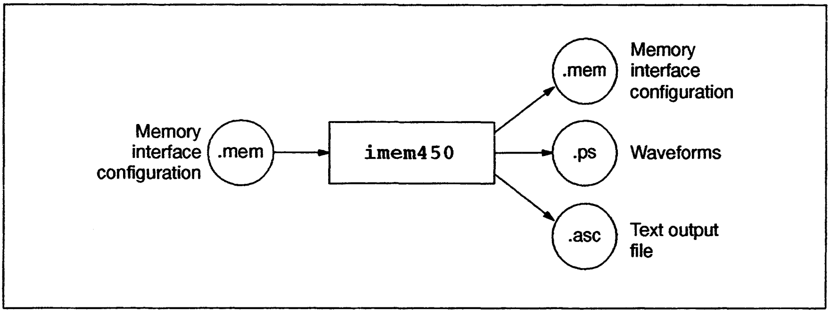

Memory interface configuration
.mem
imem450
.mem
Memory interface configuration
.ps
Waveforms
.asc
Text output file

Figure 14.1 `imem450` input and output files

14.2 Running the memory interface configurer

To invoke the memory interface configurer, the following command line is used:

► `imem450` {*options*}

where: *options* is a list of options. The options for interactive mode are given in table 14.1. The options for batch mode are given in table 14.2.

> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.
>
> Options may be supplied in an indirect argument file, prefixed by '@'. See section A.1.2 for details.

If no arguments are given on the command line a help page is displayed giving the command syntax and options.

Option	Description
`-E`	Run in interactive mode.
`-DW`	Disable warnings.
`-F` *memfile*	Specify the initial state *memfile.*
`-I`	Select verbose mode which displays status information as it runs – e.g. listing the files it opens.

Table 14.1 `imem450` command line interactive mode options

If no input file is specified in interactive mode, then an initial default configuration will be used, with refresh disabled and all the banks disabled.

Option	Description
–A	Produce an ASCII display page output file.
–DW	Disable warnings.
–F *memfile*	Specify the input *memfile.*
–I	Select verbose mode. In this mode the tool will display status information as it runs – e.g. listing the files it opens.
–O *filename*	Specify the output Postscript or ASCII filename.
–P	Produce a PostScript waveform output file.

Table 14.2 imem450 command line batch mode options

In batch mode, option –F is mandatory and one of –P or –A must be used. If the –O option is not used then an output filename will be constructed from the input filename, with an extension of .ps for a Postscript output or .asc for an ASCII output.

Examples:

The following command runs imem450 interactively, starting with the default configuration:

```
imem450 –e
```

The following command runs imem450 in verbose batch mode to produce a waveform PostScript output file:

```
imem450 –i –p –f myconfig.mem –o diagram.ps
```

14.2.1 Default command line

A set of default command line options can be defined for the tool using the IMEM450ARG environment variable. Options must be specified using the syntax required by the command line. Environment options are interpreted before other arguments. For example, to make imem450 run in verbose interactive mode by default with initial memory configuration defined by default.mem:

```
setenv IMEM450ARG "–e –i –f default.mem"
                                        (for UNIX toolsets using C shell)

set IMEM450ARG="–e –i –f default.mem"         (For MS-DOS toolsets)
```

14.3 Interactive operation

The tool's interactive user interface is presented as a number of ASCII display pages showing menus, timing data and waveforms. The timing data appears on pages 2 to 7. It may be updated interactively by entering new timing parameters from the keyboard. All inputs are executed immediately so that the user can see the effect on any of the displays. As each page is shown, the user has the option of selecting another page for display by keying in its number. The current configuration may be saved at any time to a specified output file.

The meanings of the variables on the input pages are the same as the corresponding variables in a *memfile*. The *memfile* variables are fully described in Appendix E.

Command	Action
<Return>	Go to next page
PageNumber	Go to given page
<	Scroll timing display window left
>	Scroll timing display window right
^	Scroll timing display window up
v	Scroll timing display window down
A	Generate ASCII page output file
AS	Generate ASCII sequence output file, large page
C	Change configuration data
DW	Disable warnings
E	Enter configuration data
EW	Enable warnings
H	Go to help page
L	Load memory configuration from a file
P	Generate PostScript waveform output file
PS	Generate PostScript sequence waveform output file
Q	Exit program
S	Save memory configuration to a file
SA	Sequence – append cycle
SC	Sequence – clear all entries
SD	Sequence – delete last entry
W	Display configuration warning messages

Table 14.3 Interactive commands

The ASCII display uses the file referred to by the `ITERM` environment variable. If `ITERM` is not defined then the screen will be treated as having 24 lines of 80 characters each and default character values and routines will be used. For example, the screen will be cleared by writing 24 line feeds.

The `ITERM` file can be used to improve the display. For example, the screen size can be changed to show more of the waveform diagrams. The screen size is given by the line beginning `2:` in the screen section. It can be increased to 170 columns and 50 lines by the following line:

```
    2:170,50.
```

Full details of ITERM files are given in Appendix I.

14.3.1 Interactive commands

Table 14.3 lists the interactive commands and their actions.

14.3.2 Interactive pages

Table 14.4 lists the interactive pages which may be selected. Pages 0 and 1 are introductory pages. Pages 2 to 8 are input pages, in which the user may enter or change parameters via the keyboard. Pages 9 to 19 display the effects of the parameters. Details can be entered or parameters changed while any page is displayed.

Page	Title
0	Pages menu
1	Interactive commands menu
2	Input Data, General
3	Input Data, Pad drive strengths
4	Input Data, Bank 0
5	Input Data, Bank 1
6	Input Data, Bank 2
7	Input Data, Bank 3
8	Input Data, Sequence of cycles definition
9	Register Values
10	Timing Diagram, Refresh Cycle
11	Timing Diagram, Bank 0 read cycle
12	Timing Diagram, Bank 0 write cycle
13	Timing Diagram, Bank 1 read cycle
14	Timing Diagram, Bank 1 write cycle
15	Timing Diagram, Bank 2 read cycle
16	Timing Diagram, Bank 2 write cycle
17	Timing Diagram, Bank 3 read cycle
18	Timing Diagram, Bank 3 write cycle
19	Timing Diagram, Sequence of cycles

Table 14.4 imem450 interactive pages

Page 0

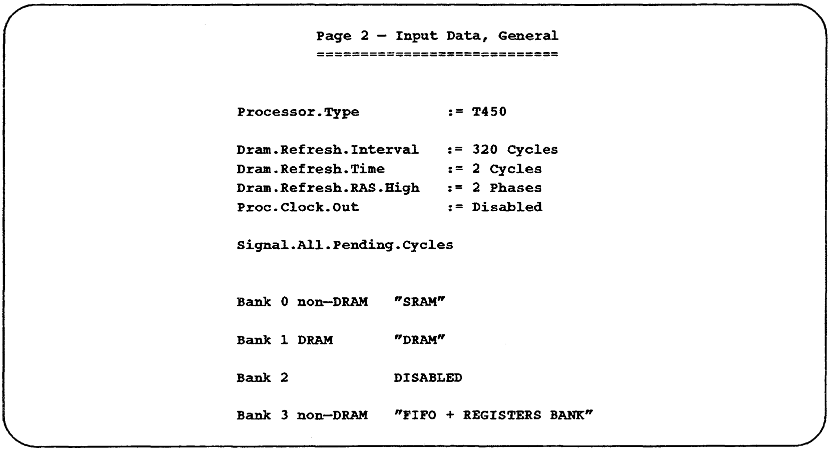
Page 0 of the imem450 display gives a list of the ASCII input and display pages, as listed in table 14.4.

Page 1

Page 1 of the imem450 display gives a list of all the commands that may be used in interactive mode, as listed in table 14.3.

Page 2

Page 2 of the `imem450` display is the input page for general data that does not relate to a particular memory bank. An example is shown in figure 14.2.

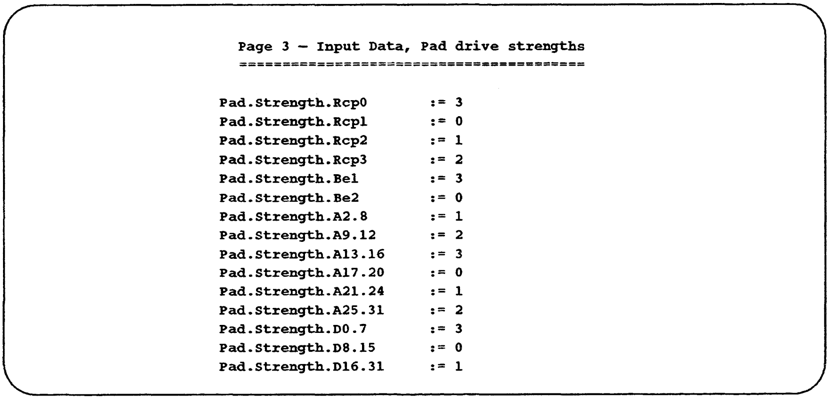


```
                    Page 2 - Input Data, General
                    ============================


           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 DRAM        "DRAM"

           Bank 2             DISABLED

           Bank 3 non-DRAM    "FIFO + REGISTERS BANK"
```

Figure 14.2 Example interactive display page 2

Page 3

Page 3 of the `imem450` display is the input page for the details of pad drive strengths. Figure 14.3 shows an example of page 3.

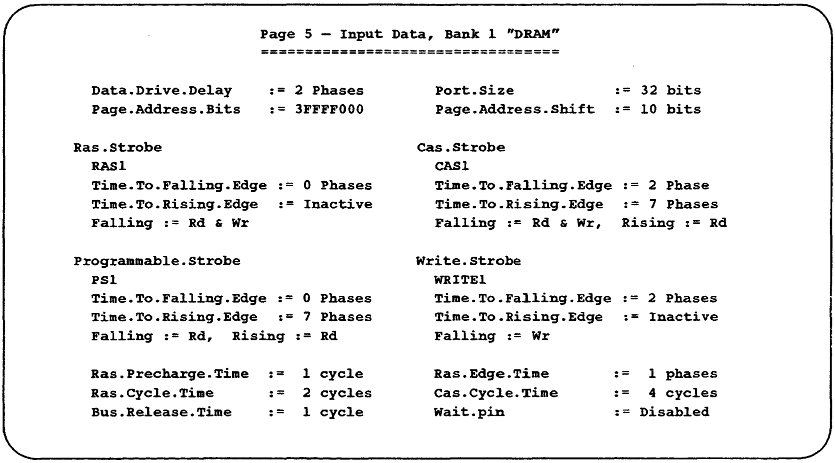


```
           Page 3 - Input Data, Pad drive strengths
           ========================================

         Pad.Strength.Rcp0       := 3
         Pad.Strength.Rcp1       := 0
         Pad.Strength.Rcp2       := 1
         Pad.Strength.Rcp3       := 2
         Pad.Strength.Be1        := 3
         Pad.Strength.Be2        := 0
         Pad.Strength.A2.8       := 1
         Pad.Strength.A9.12      := 2
         Pad.Strength.A13.16     := 3
         Pad.Strength.A17.20     := 0
         Pad.Strength.A21.24     := 1
         Pad.Strength.A25.31     := 2
         Pad.Strength.D0.7       := 3
         Pad.Strength.D8.15      := 0
         Pad.Strength.D16.31     := 1
```

Figure 14.3 Example interactive display page 3

Pages 4 to 7

Pages 4 to 7 of the **imem450** display are the input pages for the details of memory banks 0 to 3 respectively. Figure 14.4 shows an example of page 5.

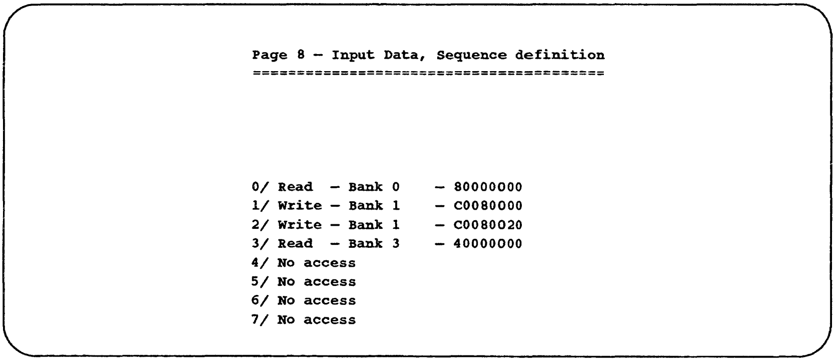


```
                         Page 5 – Input Data, Bank 1 "DRAM"
                         ==================================

    Data.Drive.Delay    := 2 Phases          Port.Size           := 32 bits
    Page.Address.Bits   := 3FFFF000          Page.Address.Shift  := 10 bits

  Ras.Strobe                               Cas.Strobe
    RAS1                                     CAS1
    Time.To.Falling.Edge := 0 Phases         Time.To.Falling.Edge := 2 Phase
    Time.To.Rising.Edge  := Inactive         Time.To.Rising.Edge  := 7 Phases
    Falling := Rd & Wr                       Falling := Rd & Wr,  Rising := Rd

  Programmable.Strobe                      Write.Strobe
    PS1                                      WRITE1
    Time.To.Falling.Edge := 0 Phases         Time.To.Falling.Edge := 2 Phases
    Time.To.Rising.Edge  := 7 Phases         Time.To.Rising.Edge  := Inactive
    Falling := Rd,  Rising := Rd             Falling := Wr

    Ras.Precharge.Time  :=  1 cycle          Ras.Edge.Time        :=  1 phases
    Ras.Cycle.Time      :=  2 cycles         Cas.Cycle.Time       :=  4 cycles
    Bus.Release.Time    :=  1 cycle          Wait.pin             := Disabled
```

Figure 14.4 Example interactive display page 5

Page 8

Page 8 of the **imem450** display is the input page to define a sequence of memory cycles for display on page 19. This facility is useful for checking precharge and bus release times and page mode for multiple memory accesses. Figure 14.5 shows an example of page 8.

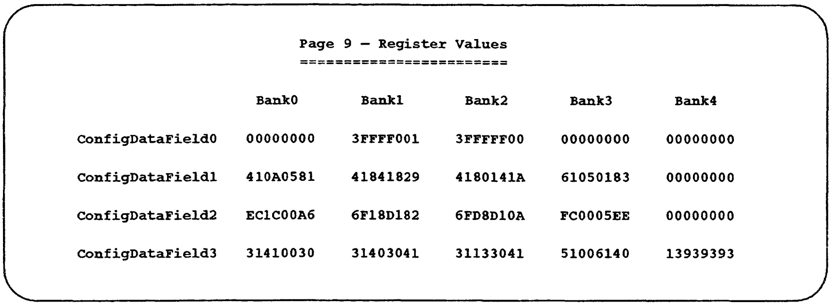


```
                        Page 8 – Input Data, Sequence definition
                        ========================================




                        0/ Read  – Bank 0    – 80000000
                        1/ Write – Bank 1    – C0080000
                        2/ Write – Bank 1    – C0080020
                        3/ Read  – Bank 3    – 40000000
                        4/ No access
                        5/ No access
                        6/ No access
                        7/ No access
```

Figure 14.5 Example interactive display page 8

Page 9

Page 9 of the `imem450` display shows the memory interface configuration register values needed to specify the parameters listed on the previous pages. Figure 14.6 shows an example.

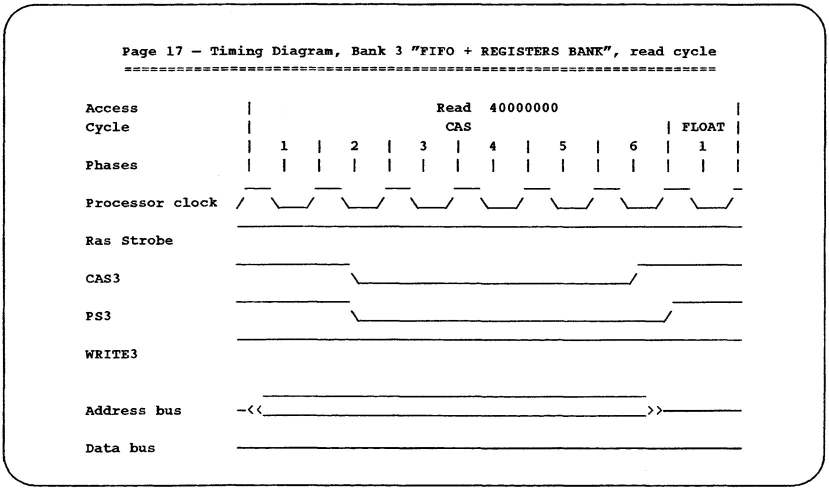


```
                           Page 9 - Register Values
                           ========================

                     Bank0       Bank1       Bank2       Bank3       Bank4

ConfigDataField0    00000000    3FFFF001    3FFFFF00    00000000    00000000

ConfigDataField1    410A0581    41841829    4180141A    61050183    00000000

ConfigDataField2    EC1C00A6    6F18D182    6FD8D10A    FC0005EE    00000000

ConfigDataField3    31410030    31403041    31133041    51006140    13939393
```

Figure 14.6 Example interactive display page 9

Pages 10 to 19

Pages 10 to 19 of the `imem450` display are the display pages for the waveform timing diagrams. Page 10 shows the refresh cycle. Pages 11 to 18 show the read and write cycles of memory banks 0 to 3. Page 19 shows the sequence of read, write and refresh cycles defined on page 8. Figure 14.7 shows an example of page 17.



```
   Page 17 - Timing Diagram, Bank 3 "FIFO + REGISTERS BANK", read cycle
   ====================================================================

Access            |                Read  40000000                  |
Cycle             |                  CAS                   | FLOAT |
                  |  1  |  2  |  3  |  4  |  5  |  6  |  1  |
Phases            |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
                    __    __    __    __    __    __    __    _
Processor clock  /  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/
                 __________________________________________________
Ras Strobe
                 ___________                         ______________
CAS3                        \_______________________/
                 ___________                            ___________
PS3                         \__________________________/
                 __________________________________________________
WRITE3
                     ___________________________________
Address bus      -<<___________________________________>>----------

Data bus         __________________________________________________
```

Figure 14.7 Example interactive display page 17

14.4 Output files

Three different types of output may be produced by **imem450**, as listed below:

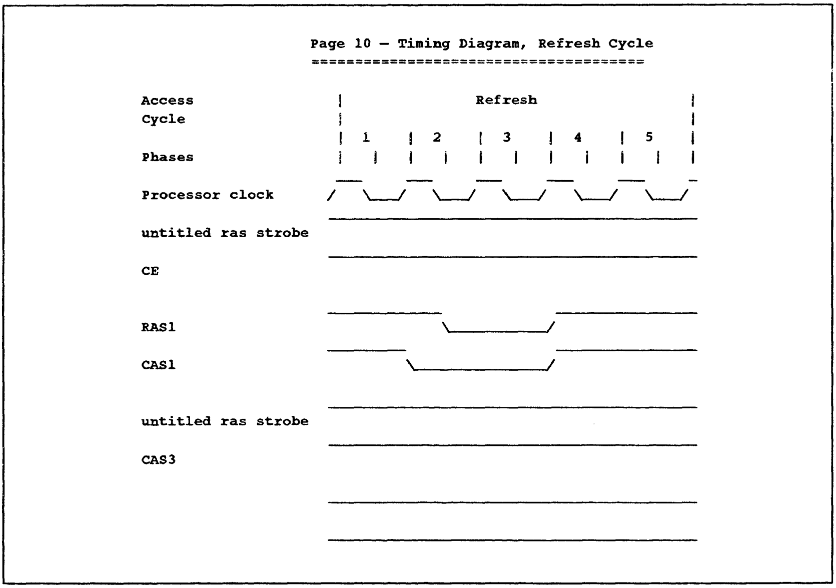
- A *memfile* suitable for including as an input file to the initialization file generator or the EPROM program formatting tool or for further editing using **imem450**.
- An ASCII display page output file, suitable for inclusion in documentation.
- A waveform output file in Postscript format, suitable for inclusion in documentation.

14.4.1 Memfiles

The **imem450** tool can be used interactively to create or modify a memory interface configuration file, or *memfile*. **imem450** is capable of saving and reloading *memfiles* to allow for design and modification over an extended period and for comparison of different configurations. The *memfile* is described and an example is given in Appendix E.

Memfiles are simple text files that may also be created or modified manually using a standard text editor.

The memfile is read during the building of the binary code file and the configuration information is either incorporated into the bootstrap for booting from a host or into a ROM.



```
                        Page 10 - Timing Diagram, Refresh Cycle
                        =======================================

Access                  |                 Refresh                   |
Cycle                   |                                           |
                        |  1     |  2     |  3     |  4     |  5    |
Phases                  |    |   |    |   |    |   |    |   |   |   |
                           ___     ___     ___     ___     ___     _
Processor clock         /     \___/   \___/   \___/   \___/   \___/
                        ____________________________________________
untitled ras strobe
                        ____________________________________________
CE

                        ______________              ________________
RAS1                                  \____________/
                        __________                  ________________
CAS1                                \______________/
                        ____________________________________________
untitled ras strobe
                        ____________________________________________
CAS3

                        ____________________________________________

                        ____________________________________________
```

Figure 14.8 A page of an example ASCII display page output file

To create, modify or view a *memfile*, the `-E` run-time option should be selected.

14.4.2 ASCII display page output files

The `imem450` tool may be used to produce timing data and waveforms in the form of ASCII text output files. These files can be printed and included in reports or other documentation.

To produce an ASCII display page output file in batch mode, the `-A` run-time option should be selected. An ASCII page output file may also be produced in interactive mode with the commands `A` and `AS`. The file produced by the `-A` run-time option or `A` interactive command will consist of all the pages which could be displayed in interactive mode, as described in section 14.3.2. The `AS` command produces a file containing just the full size sequence page, page 19. Part of an example of an output file is shown in figure 14.8.

The output is an ASCII text file which may be edited using an ordinary text editor.

14.4.3 Waveform file

The `imem450` tool may be used to produce waveform data in the form of a PostScript format output file. This file can be printed and included in reports or other documentation. The diagrams provided are the refresh cycle and the read and write cycles for all active banks.

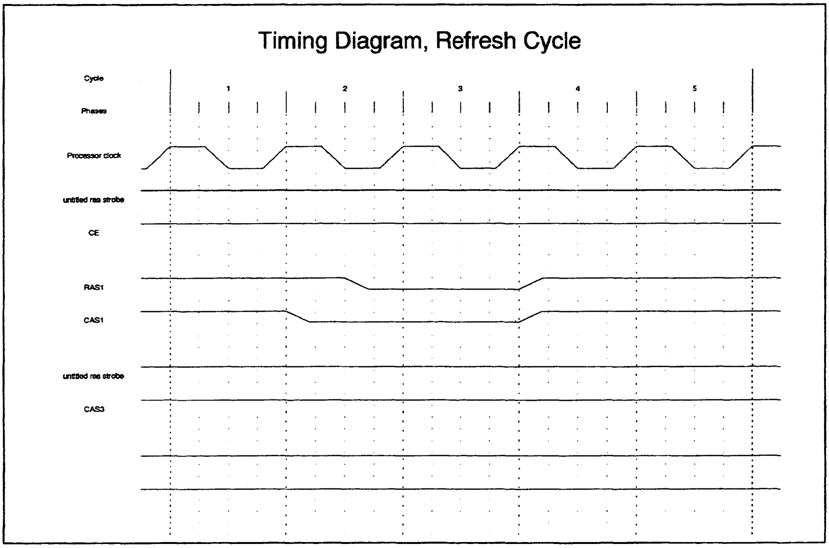


Figure 14.9 Refresh cycle waveform postscript output example

To produce a waveform diagram file in batch mode, the `-P` run-time option should be selected. In interactive mode, a waveform diagram file may be produced with the command P or PS. The file produced by the –P run-time option or P interactive command will consist of PostScript versions of the waveform diagrams for refresh and for read and write for the enabled banks. Example pages are shown in figures 14.9 and 14.10.

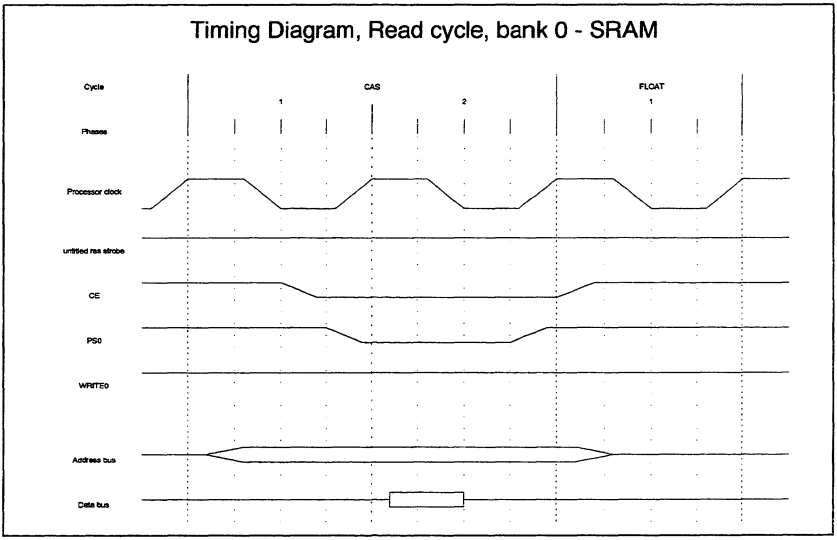


Figure 14.10 Read cycle waveform postscript output example

The PS command produces a file containing a PostScript version of the waveform diagram for the sequence of cycles, as displayed on page 19. An example is shown in figure 14.11.

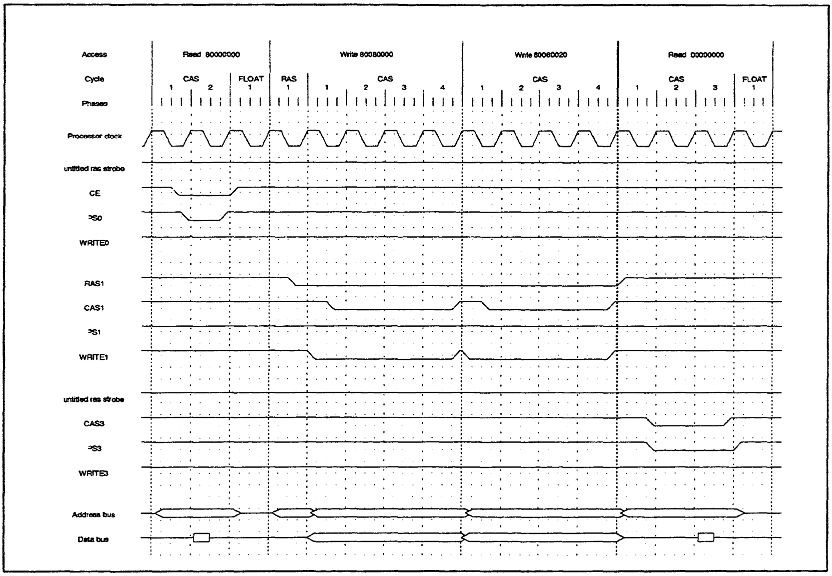


Figure 14.11 Postscript output example of a sequence of cycles

14.5 Error messages

This section lists each error message that can be generated by `imem450`. Messages are in the standard toolset format which is explained in Appendix A.7.

14.5.1 Warnings

bank *number* CAS strobe has rising edge before falling edge.

During a cycle, an active strobe will start high. When the user specified '`Time.To.Falling.Edge`' has elapsed, it will go low, and will remain low until '`Time.To.Rising.Edge`', or the end of the cycle (if '`Time.To.Rising.Edge`' is not specified). Both of these times are specified from the start of the cycle. This message indicates that the user requested the strobe go back to high, before it went low.

bank *number* no strobes enabled for this bank.

Self explanatory, all the strobes have been defined to be inactive during accesses, the outside world won't even know that an access has been made to the bank.

bank *number* PAGE.ADDRESS.SHIFT is too large, not all page address bits will be on the bus during RAS cycle.

During a RAS cycle, when the page address is presented, the address bus is shifted down by an amount specified by the user. i.e. if the page address bits mask is

000FFC00 then it is likely that the address will be shifted down by, for example, 8 to present the address on bits 2..11. If it was shifted by more than 10, then some page address bits will be shifted out, and will not be presented on the bus during the RAS cycle.

bank *number* Programmable strobe has rising edge before falling edge.

During a cycle, an active strobe will start high. When the user specified '`Time.To.Falling.Edge`' has elapsed, it will go low, and will remain low until '`Time.To.Rising.Edge`', or the end of the cycle (if '`Time.To.Rising.Edge`' is not specified). Both of these times are specified from the start of the cycle. This message indicates that the user requested the strobe go back to high, before it went low.

bank *number* RAS.EDGE.TIME is longer that the RAS.CYCLE.TIME

During a RAS cycle, the RAS strobe stays high for a time then goes low until at least until the end of the RAS cycle. The `RAS.EDGE.TIME` specifies how long the strobe stays high. If this time is longer than the cycle time then the RAS strobe will not go low during the RAS cycle.

bank *number* Strobes extend beyond end of CAS cycle time.

During a cycle, an active strobe will start high. When the user specified '`Time.To.Falling.Edge`' has elapsed, it will go low, and will remain low until '`Time.To.Rising.Edge`', or the end of the cycle (if '`Time.To.Rising.Edge`' is not specified). Both of these times are specified from the start of the cycle. This message indicates that one of these times, specified by the user, was greater than the entire cycle time, specified as `Cycle.Time` or `Cas.Cycle.Time` depending on whether or not this is a DRAM.

bank *number* Write strobe has rising edge before falling edge.

During a cycle, an active strobe will start high. When the user specified '`Time.To.Falling.Edge`' has elapsed, it will go low, and will remain low until '`Time.To.Rising.Edge`', or the end of the cycle (if '`Time.To.Rising.Edge`' is not specified). Both of these times are specified from the start of the cycle. This message indicates that the user requested the strobe go back to high, before it went low.

14.5.2 Errors

These errors may occur during the processing of the memory configuration file and are considered to be self explanatory:

Attempt to define bank *number* twice.

Attempt to specify *string* twice.

Bank number, *number*, out of range, expecting 0..3

`BUS.RELEASE.TIME` **must be specified.**

`CAS.CYCLE.TIME` **must be specified.**

Decimal number *number* to large – *string*

Expecting '"' to open string – *string*.

Expecting assignment operator ':='

First statement in bank definition must be `device.type`

Hexadecimal number too large – *string*

If '`inactive`**' is used it must be the only statement in a strobe definition**

Illegal decimal number – *string*

Illegal hexadecimal number – *string*

Legal options are :– <*options*>

Missing decimal number

Missing hexadecimal number

Missing keyword
Legal options are :– <*options*>

Missing name

Missing string

Missing word, expecting '*string*'

Number, *number*, out of range, expecting 0..15

Number, *number*, out of range, expecting 0..1023

`PAGE.ADDRESS.BITS` **must be specified for a DRAM.**

`RAS.CYCLE.TIME` **must be specified for a DRAM bank.**

`RAS.EDGE.TIME` **must be specified for a DRAM bank.**

`RAS.PRECHARGE.TIME` **must be specified for a DRAM bank.**

`RAS.STROBE` **must be inactive or have falling edge specified in DRAM bank.**

Statement has no meaning in non–DRAM bank.

Statement must occur inside bank definition

Statement must occur inside strobe definition

Statement must occur outside bank definition

Statement must occur outside strobe definition

Status of wait pin not specified.

Surplus data after valid statement – '*string*'

Time out of range, expecting 2..6 phases, or 1..3 cycles

Time out of range, expecting 0..30 phases, or 0..15 cycles

Time out of range, expecting 2..30 phases, or 1..15 cycles

Time out of range, expecting 4..30 phases, or 2..15 cycles

Time out of range, expecting 0..3 phases

Time out of range, expecting 0..7 phases

Time out of range, expecting 0..31 phases

Time to edge out of range, expecting 0..63 phases, or 0..15 cycles

Unable to open memory configuration file '*string*'

Unable to open output file

Unrecognised keyword '*string*'

Unrecognised statement keyword – *string*

Unrecognised word '*string*', expecting '*string*'

Unterminated string – "*string*

Value must be whole number of cycles, or number of phases divisible by 2.

Value uses bits 0 & 1, illegaly

14.5.3 Fatal errors

This version of sparse does not support activity *number*

> An internal error has occurred. This should be reported to your local SGS-THOMSON distributor of field applications engineer.

15 irun - application loader

This chapter describes how to use irun to load and run an application on the target hardware. Applications may either use iserver protocols or the AServer or a combination of both. irun also fully supports the INQUEST tools. The INQUEST tools have their own commands which in turn load irun in order to load the application.

15.1 The purpose of irun

irun performs three functions, namely:

1. to initialize the target hardware;
2. to load a bootable application program onto the target hardware;
3. to serve the application, i.e. to respond to requests from the application program for access to host services, such as host files and terminal input and output.

These steps are normally performed when irun is invoked, and are described in more detail below.

15.1.1 Initializing target hardware

Normally, the target hardware must be initialized before an application can be loaded onto it. For hosted systems, including development systems, this is performed automatically by irun in combination with any boot ROM.

15.1.2 Loading programs

Before an application program can be loaded onto target hardware, a bootable file must be built, as described in the '*User Guide*'. By convention, a bootable file generally has a .btl file extension. The process of building includes compiling, linking, configuring and collecting.

Bootable files contain a loader and the application code. If the target hardware has been initialized, then the bootable file may be sent to the target. This will cause the target to be bootstrapped and the application to be loaded and start running.

15.1.3 Access to host services

Once an application is loaded and running, it can communicate with the host using the i/o libraries for the language being used. The library calls available and their parameters are documented in the '*Language and Libraries Reference Manual*'.

irun supports the iserver protocol which is used by the ANSI C and occam 2 run-time libraries to communicate with the host.

15.2 Starting an application

An application is started on the target hardware by the `irun` command. The `irun` command can be entered at a SunOS prompt. For Microsoft Windows users, the `irun` command can be entered in a DOS window provided `ilaunch` is running. For Windows users, the `irun` command may also be implemented by the normal Windows methods, as described in section 15.2.3.

15.2.1 Target interface parameters

Before starting `irun`, the target interface parameters must be set up. `irun` uses two parameters, `TRANSPUTER` and `ASERVDB`, which should have been set up during installation of the software, although they may need to be changed from time to time. These are described in more detail in section 15.3. A launch tool and an `iset` tool are provided to facilitate setting these parameters from Microsoft Windows. The launch tool is described in Chapter 8 and the `iset` tool is described in Chapter 16.

Target interface parameters may be stored as environment variables or for Microsoft Windows they may be stored in a Windows environment file. The value of `TRANSPUTER` may also be given at run time as a command line option. Values stored in a Windows environment file override DOS environment variables but do not change them. Command line options override any Windows environment file and any environment variables but do not change them.

15.2.2 The `irun` command line

`irun` can be started using the following command line:

▶ `irun` {*options*} {*bootable_file*} {*options*}

where: *bootable_file* is the name of the application code bootable file,

options is a list of one or more options from table 15.1.

> Options may be entered in upper or lower case.
>
> Options can be given in any order and the order may be significant.
>
> Options must be separated by spaces.

If `irun` is invoked with no options and no bootable file, then help information is displayed, briefly explaining the command line arguments.

The full list of command line options for this release of `irun` is given in table 15.1.

Option	Description
–NE	Turn off monitoring for errors while serving the link.
–NR	Turn off resetting the network before booting.
–SAD *entry*	Add *entry* to the AServer database for this execution of the program.
–SB *bootable_file*	Load *bootable_file* and serve the link, monitoring for errors. This is the same as specifying *bootable_file* as the parameter. This option is provided for compatibility with earlier versions.
–SC *bootable_file*	Load *bootable_file* onto the target hardware.
–SCL *command*	Start the client or service with command *command.*
–SE	Turn on monitoring for errors while serving the link.
–SI	Turn on display of progress information as the program is loaded.
–SL *resource*	Use the target hardware connection *resource*, which is the name of a resource in the AServer database. This overrides the TRANSPUTER environment variable.
–SN *filename*	Initialize an IMS T9000 network using the named .nif file. This overrides the AServer database entry.
–SR	Reset the target hardware.
–SS	Serve the link without resetting, i.e. respond to host i/o requests from the target (not IMS T9000s unless a bootable file is given).
–ST	Pass any following arguments to the application.

Table 15.1 irun command line options

For example, to load the application hello.btl, enter the command:

```
irun hello.btl
```

To load the application hello.btl onto the target hardware b008, enter the command:

```
irun –sl b008 hello.btl
```

The order of some of the irun options in the command line is significant, so that several bootable files may be loaded in succession with a single irun command. Each irun option implies one or more actions by irun and the actions are carried out in the following order:

1. all actions except loading and serving;
2. load all bootable files given with the SC option in the order they are given in the command line;
3. load any bootable file given as a parameter or with the SB option;
4. serve the link if required (i.e. if the SS or SB option is given or a bootable file without an option is given).

15.2.3 Starting using Microsoft Windows

irun can be run in the same way as other Windows applications. Four possible methods are:

- Use the Program Manager to create a program group and assign the command line given in section 15.2.2 to an appropriate icon. New icons can be created using **New...** in the **File** menu. The command line assigned to an icon can be changed by selecting the icon and using **Properties...** in the **File** menu.
- Use the File Manager to associate `irun` with the extension `.btl`. In this case, no options are permitted. Double clicking on a `.btl` file in the File Manager will then start `irun`.
- Click on `irun.exe` in the File Manager. This will open a Command Line Box, with fields for the bootable file to be run, the options and the working directory. The possible options are listed in table 15.1. The File field has a **Browse** button and the Options field has a history button.
- Select the **Run...** command in the **File** menu of the File Manager and enter the command line, as given in section 15.2.2.

If `ilaunch` is running then `irun` commands may be entered in a DOS window.

15.3 The environment

`irun` uses three environment parameters, `ISEARCH`, `ASERVDB` and `TRANSPUTER`, which should have been set up during installation of the software, although they may need to be changed from time to time. `irun` needs to communicate with the target hardware, and uses these parameters to identify which connection is to be used and the method of communication.

Parameter	Meaning
`ISEARCH`	Search path for libraries and include files.
`ASERVDB`	Pathname of AServer database file if not on the `ISEARCH` path.
`TRANSPUTER`	Name of target hardware connection.

Table 15.2 Environment parameters

15.3.1 `ISEARCH`

The `ISEARCH` parameter gives the path which is searched by certain tools for libraries, include files and other files. Each directory in the path should be terminated with a path separator (\ on PCs, / on Suns) as the directory is directly prepended to the filename. Directories are separated by spaces.

15.3.2 `ASERVDB`

The `ASERVDB` parameter gives the pathname of the AServer database file if the file is not on the `ISEARCH` path. The AServer database defines possible resources, which may be target hardware connections or host processes which may be requested by the application. For each hardware connection the AServer database gives the method of communication. The AServer database file is described in Appendix H.

15.3.3 TRANSPUTER

The TRANSPUTER parameter specifies which target hardware connection is to be used. The possible values are the resource names in the AServer database pointed to by ASERVDB. The AServer database is described in Appendix H.

irun uses the TRANSPUTER parameter to find the connection to serve and to which bootable files are to be copied. The TRANSPUTER variable may be overridden by the SL option on the irun command line.

15.3.4 Setting target interface parameters on a Sun

On a Sun, the target interface parameters are environment variables, and may be set in the same way as other environment variables.

15.3.5 Setting target interface parameters on a PC with Windows

On a PC with Windows, the target interface parameters should normally be set using the Windows launch tool ilaunch, which is described in Chapter 8 or the iset command line tool, which is described in Chapter 16. These tools are used to set values in the Windows environment file.

The Windows launch tool is normally started automatically when Windows is started and allowed to run throughout the Windows session, represented by a Windows icon appearing at the bottom of the screen. The iset tool is run by a command line in a DOS window and sets or clears a single parameter.

15.4 Skip loaders

This section describes the skip loaders, skip*n*.btl. The skip loaders allow code to be loaded onto and run on sub-networks of a transputer network which do not include the root transputer, i.e. it allows one or more transputers to be skipped. One or more skip loaders may be loaded before loading the final code. The skip bootables are not included with ST20 toolsets.

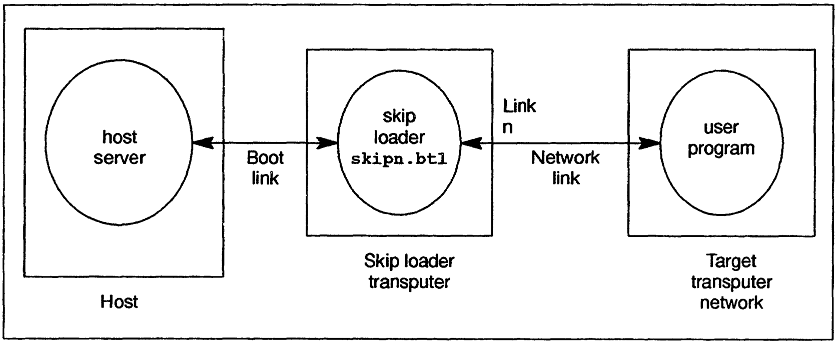


Figure 15.1 Using one skip loader

15.4 Skip loaders

A skip loader copies all data received on the boot link to a specified network link and copies all data received from the specified network link to the boot link, as shown in figure 15.1. Thus the transputer on which it runs becomes effectively transparent so that code can be loaded through it and messages may be passed through it to and from the host as if there were a direct connection. The boot link is the link from which the skip transputer was booted. The network link is given by the particular skip loader used.

There are four skip loaders, as follows:-

- `skip0.btl` Network link is link 0 of the skip loader transputer;
- `skip1.btl` Network link is link 1 of the skip loader transputer;
- `skip2.btl` Network link is link 2 of the skip loader transputer;
- `skip3.btl` Network link is link 3 of the skip loader transputer.

Multiple skip loaders may be used to skip as many transputers as necessary. The chain of boot links then forms a path from the host to the user application.

Resetting the transputer network after the skip loader has been loaded will cause the skip loader to be lost. The network can be reset using the `irun` option `SR`. An application to be skip loaded must be loaded using the `NR` option or the skip loader will be lost.

15.4.1 Running the skip loader

To load a single skip loader using `irun` use the following command line:

```
irun —sr —sc skipn.btl
```

where: *n* is the number of the link from the skip loader to the target network.

15.4.2 Examples of use

The following example 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 root transputer:

```
irun -sr -sc skip2.btl -sc example.btl -se -ss
```

The following example saves the extracted results from `example.btl` which was running on the network found down link 2 from the root transputer. It then displays those results.

```
iprof -sc skip2.btl example.btl
```

16 iset - Windows parameter tool

The purpose of the iset Windows parameter tool is to set and clear tool parameters defined in the Windows environment file from DOS.

The application loader irun uses the parameters described in Chapter 15. On PC systems using Microsoft Windows, these parameters may be held either as DOS environment variables or in a Windows environment file or both. The Windows parameter tool iset is used to edit a line of the Windows environment file from a DOS command line or in a batch file, such as autoexec.bat. iset may only be used on a PC with Microsoft Windows.

The Windows environment file is described in Chapter 8.

The iset command has the syntax:

► iset *command*

where: *command* is a string describing the action required, which must be in one of the following forms:

- *variable = value*

  This writes the following assignment statement into the ENVIRONMENT section of the Windows environment file:

  variable = value

- *environment_variable*

  This writes the following assignment statement into the ENVIRONMENT section of the Windows environment file:

  environment_variable = value

  where *environment_variable* is a DOS environment variable with the value *value*. This effectively copies the environment variable into the Windows environment file.

- *variable =*

  This deletes from the Windows environment file any assignment statement to *variable*.

Spaces are not permitted on either side of the equals sign (=).

Examples

The following command line sets the parameter ISEARCH to search C:\myapp and C:\toolset\libs by writing the corresponding assignment statement in the Windows environment file:

```
iset ISEARCH=C:\myapp\ C:\toolset\libs\
```

The following command line sets the parameter TRANSPUTER to the value of the TRANSPUTER environment variable by writing the corresponding assignment statement in the Windows environment file:

```
iset TRANSPUTER
```

The following command line deletes the parameter ASERVDB from the Windows environment file, which means that tools needing an AServer database will use the one defined in the ASERVDB environment variable:

```
iset ASERVDB=
```

Appendices

A Toolset conventions and defaults

This appendix describes the standards and conventions used by the toolsets for:

- Command line syntax
- Filenames
- Search paths
- File extensions
- Error message format.

A.1 Command line syntax

All tools in the toolsets conform to a common command line format.

A.1.1 General conventions

- Commands, and their parameters and options, obey host system standards.
- Filenames, either directly specified on the command line or as arguments to options, must conform to the host system naming conventions.
- Indirect argument files are prefixed by the character '@', see section A.1.2.
- Options must be prefixed with the option prefix character '–'. (**Note:** that MS-DOS based toolsets also allow the use of '/' as a prefix character).
- Command line parameters and options can be specified in any order but must be separated by spaces.
- If no parameters or options are specified the tool displays a help page that explains the command syntax.

A.1.2 Indirect argument files

Arguments to a command line can be supplied in an indirect file introduced by the prefix character '@'. This enables host system restrictions on command line length to be avoided. Files prefixed by '@' are searched for within the current directory. If the filename is relative then it is considered to be relative to the current directory.

An indirect argument file is treated as a text file containing a sequence of command lines separated by newline characters. Each line is treated as a single command line, arguments being separated by tabs or blank spaces. Blank spaces can be included in an argument by surrounding the argument string by double quotes "". If a double quote is to be contained within the argument string then the double quote should be repeated. e.g.

```
"string with ""quotes"" inside"
```

converts to:

```
string with "quotes" inside
```

Each line in the file is treated as being complete in itself, i.e. a prefix character that expects an operand, must have its operand on the same line. All string arguments surrounded by double quotes must have the terminating double quote character on the same line.

Lines beginning with a hash symbol '#' are treated as comments and are ignored.

Example

The ANSI C compiler can be invoked with the command:

```
icc @localopt.txt prog.c
```

where: 'localopt.txt' may contain the line (for MS–DOS based systems):

```
/t8 "/DHOST=""MS DOS"""
```

which will define the macro HOST to have the value "MS DOS" including the quotes.

'prog.c' is the name of the source file to be compiled.

A.2 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.

A.3 Filenames

File names generally follow the naming and character set conventions of the host operating system except that the following directory separator characters cannot be used within a filename:

- Colon ':'
- Semi–colon ';'
- Forward slash '/'

- Backslash '\' ('¥' for Japanese MS-DOS)
- Square brackets '[]'
- Round brackets '()'
- Angle brackets '<>'
- Exclamation mark '!'
- Equals sign '='

Filenames prefixed by the character '@' indicate an indirect argument file, see section A.1.2.

A.4 Search paths

The tools locate files by searching a specified *directory path* on the host system. The path is specified using the host environment variable `ISEARCH`. The search rules for all tools except compilers are listed below:

(See the compiler reference chapters for details of search rules for compilers).

1. If the filename contains a directory specification then the filename is used as given. Relative directory names are treated as relative to the directory in which the tool is invoked.
2. If no directory is specified the directory in which the tool is invoked is assumed.
3. If the file is not present in the current directory, the path specified by the environment variable (or logical name) `ISEARCH` is searched. If there are several files of the same name on this path, the first occurrence is used.
4. If the file is not found using the above rules, then the file is assumed to be absent, and an error is reported.

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

By default all files are written to the current directory.

A.5 Standard file extensions

The toolsets use a standard set of file extensions for source and object files. In most cases these extensions must be specified on the command line for input files. They are automatically created for output files, unless an alternative filename is specified on the command line.

A separate set of extensions for object files must be used where `imakef` is used to build programs for mixed processor networks. These are described separately in section A.6.

A.5.1 Main source and object files

Extension	Description
.btl	Bootable file which can be loaded onto a transputer or transputer network. Created by icollect directly from a .lku file (single transputer programs) or from a .cfb file. Bootable files can be sent down a link by the server for immediate execution. Contains information used by the server to control the host link for execution. Also read by the debugger.
.c	C source files. Assumed by icc, the ANSI C compiler.
.cfb	Configuration binary file containing a description of how code is to be placed on a network, a description of the route to be used to load the network, and the parameters to be passed to each of the processes. Created by the configurer from a user-defined configuration description and read by icollect to prepare a bootable file and by the debugger.
.cfs	Configuration description file. This is a text file, created by the user and describes the hardware and software networks and the mapping between them. It also references the linked units and is used as input to the C configurer icconf.
.epr	EPROM control file. Read by ieprom.
.h	Header files for use in C source code.
.inc	Include files named in #INCLUDE compiler directives for occam, or #include statements in configuration descriptions.
.lku	Linked unit. Created by ilink as an executable process with no external references. Referenced from a configuration description. Also read by the debugger.
.lib	Library file containing a collection of binary modules. Created by ilibr.
.mem	Memory configuration file. Created and manipulated by iemit and imem450. Read by ieprom.
.occ	occam source files. Assumed by oc, the occam 2 compiler.
.pgm	occam configuration description file. This is a text file, created by the user and describes the hardware and software networks and the mapping between them. It also references the linked units and is used as input to the occam configurer occonf.
.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 binary module produced by all SGS-THOMSON TCOFF compilers. Used as input to ilink and ilibr. Also read by the debugger.

A.5.2 Indirect input files (script files)

Extension	Description
.lbb	Library build files which specify the components of a library to ilibr.
.liu	Library usage files. Created and used by imakef.
.lnk	Linker indirect files which specify the components of a program to be linked. Also used by imakef when creating Makefiles.

A.5.3 Files read by the memory map tool imap

Extension	Description
.mxx	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. For example if the compiler object file takes the default extension .tco, the information file is given the extension .mco.
.dxx	Map file output by the linker. The characters 'xx' are determined by the 2nd and 3rd characters of the extension given to the linker output file. For example if the linker output file takes the default extension .lku, the information file is given the extension .dku.
.map	Map file output by the collector.
Note: These extensions also satisfy imakef's requirements, see section A.6.	

A.5.4 Other output files

Extension	Description
.asc	ASCII format file output by iemit and imem450, showing memory configuration timings.
.bin	Binary format files produced by ieprom for loading into ROM.
.btr	Executable code without a bootstrap. Created by icollect and used as input to ieprom.
.clu	Configuration object file, created by the occam configurer occonf.
.hex	A hex dump of a file for loading onto a ROM by a custom ROM loader tool.
.ihx	Intel hex format files produced by ieprom for loading into ROM.
.mot	Motorola 'srecord' files produced by ieprom for loading into ROM.
.ps	Postscript format file output by iemit and imem450 showing memory configuration timings.
.rsc	An .rsc file contains the code of a process together with a description of its requirements for data areas and parameters. It is created by the collector from a linked unit. The format is described in chapter 5. .rsc files are suitable for using with either the occam or C functions which support dynamic code loading.

A.5.5 Miscellaneous files

Extension	Description
.itm	ITERM files containing information about the terminal. Used by tools such as iemit and imem450 to handle the screen in a device-independent manner. Can also be created by users for other terminals. The file is referenced via the environment variable ITERM.
.mak	Makefile generated by imakef. This file may be input to a make utility to build the target file. May also be edited by the user.

A.6 Extensions required for imakef

The standard set of file extensions are adequate for simple programs executing on a single transputer, or on a network of transputers all of the same type. If the network is heterogeneous and a particular source file needs to be compiled for more than one transputer type, the following scheme can be used to identify the individual processor types and error modes.

If `imakef` is used to build the program, this scheme *must* be used.

The extended system uses extensions of the form `.`*fpe*:

where: *f* denotes the type of file and can take the following values:

`t` for `.tco` equivalents.

`l` for `.lnk` equivalents.

`c` for `.lku` equivalents.

`r` for `.rsc` equivalents.

p denotes the transputer target type or class. This can take the following values:

2 – IMS T212, T222, M212

3 – IMS T225

4 – IMS T414

5 – IMS T400, T425

6 – IMS T450

8 – IMS T800

9 – IMS T801, T805

`a` – IMS T400, T414, T425, T800, T801, T805

`b` – IMS T400, T414, T425

e denotes the execution error mode. The values it can take are:

`h` – on execution, an error will immediately halt the transputer.

`s` – when an error occurs, this process will terminate.

`x` – the program can be executed in either HALT or STOP mode.

A.7 Message handling

All tools in the toolsets display diagnostic messages in a standard format. This has certain advantages:

1 The tool generating the message can be identified even when the tool is run out of contact with the terminal.

2 User programs or system utilities can be used to detect and manipulate errors. Some host system editors permit automatic location of errors.

A.7.1 Message format

Diagnostic messages are displayed in a standard format by all tools. The generalized format can be expressed as follows:

severity – *toolname* – [*filename* [(*linenumber*)]]–*message*

where: *severity* indicates the severity level. Severity categories are described below.

toolname is the standard toolset name for the tool. Names used to rename tools by the user, are not used.

filename and *linenumber* indicate the file and line where an error occurred. They are only displayed if the error occurs in a file. They are commonly displayed when files of the wrong format are specified on the command line, for example, a source file is specified where an object file is expected.

message explains the error and may recommend an action.

A.7.2 Severities

The severity attached to the message indicates the importance of the diagnostic to the operation of the tool. It also implies a certain action taken by the tool.

Five severity categories are recognized:

Information Warning Error Serious Fatal

Information messages provide the user with information about the functioning or performance of the tool. They do not indicate an error and no user action is required in response.

Warning messages identify minor logical inconsistencies in code, or warn of the impending generation of more serious errors. The tool continues to run and may produce usable output if no errors are encountered subsequently.

Error messages indicate errors from which the tool can recover in the short-term but may cause further errors to be generated which may lead to termination. The tool may continue to run but further errors are likely and the tool is likely to abort eventually. No output is produced.

Serious errors are errors from which no recovery is possible. Further processing is abandoned and the tool aborts immediately. No output is produced.

Fatal errors indicate internal inconsistencies in the software and cause immediate termination of the operation with no output. Fatal errors are unlikely to occur but if they do the fact should be reported to your local SGS–THOMSON distributor or field applications engineer.

A.7.3 Runtime errors

Errors which prevent the program from being run are detected by the runtime system at startup or during program execution. These errors are displayed in a similar format to that used by the tools. All runtime errors are generated at *Fatal* severity and cause immediate termination of the program.

B Processor types and classes

This appendix first identifies the processor type(s) supported by different toolsets. It then explains the concept of processor classes in terms of developing programs for multiple targets. This includes compiling and linking program modules. The examples given are based on the 'Hello world' program, written in C and compiled with the `icc` compiler.

It also explains the command line options which can be used to specify a target processor or class.

Note: the information given in this appendix covers a broad range of processors; readers should ignore details of processors which do not apply to their particular toolset.

B.1 Processor types supported by the toolset

The ST20 toolset can be used to develop programs targetted at the ST20 family of processors. (Because code developed for the ST20 cannot be run on other processor types discussed in this appendix, the appendix has little relevance to the current ST20 toolset).

The ANSI C and occam 2 toolsets support the following processor types:

IMS T225, T400, T425, T450, T805

and can be used to generate code for the following obsolete processors (support for which may not be included in future toolsets):

IMS M212, T212, T222, T414, T800, T801

B.2 Processor types and classes

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

The ST20 and the T450 must not be mixed with any of other processor types mentioned in this appendix, see section B.2.4.

B.2.1 Single processor type

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

Examples:

To compile and link for an ST20:

```
icc hello.c –st20
ilink hello.tco –st20 -f cstartup.lnk
```

To compile and link for a T450:

```
icc hello.c –t450
ilink hello.tco –t450 -f cstartup.lnk
```

To compile and link for a T805:

```
icc hello.c -t805
ilink hello.tco -t805 -f cstartup.lnk
```

B.2.2 Creating a program which can run on a range of processors

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

A processor class identifies an instruction set which is common to all the processors in that class, as described in section B.2.4. When a program is compiled and linked for a processor class it may be run on any member of that class.

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

Table B.1 lists all the processor classes which the compiler and linker support and indicates which processors the program can be run on.

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

Table B.1 Processor classes

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

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

2 Using Table B.1 select the class which may be run on all the target processors.

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

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

```
icc hello.c -t5
ilink hello.tco -t5 -f cstartup.lnk
```

Alternatively to create a program which will run on an IMS T400, T425 or a T805, compile and link for processor class TA:

```
icc hello.c -ta
ilink hello.tco -ta -f cstartup.lnk
```

Code compiled for an IMS T414 (class T4) may be run on an IMS T400, T414 or T425, which form class T5.

Programs compiled for the IMS T212, M212, or T222 processors i.e. class T2, can be run on a T225 (class T3) because an IMS T225 has a similar but larger instruction set than class T2 processors. Similarly the IMS T400 and T425 have additional instructions to those of the IMS T414. Likewise, code compiled for an IMS T800 (class T8) may be run on an IMS T800, T801 or T805, which form class T9. Again the IMS T801 and T805 have additional instructions to those of the IMS T800. See section B.2.4.

B.2.3 Linking files which contain code compiled for different targets

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

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

- An individual module can be compiled once e.g. for class T4, and then linked with separate programs to run on different processor types e.g. IMS T400 and T425.

- When the user is preparing a library for use by programs intended to run on different processor types, a single copy of code compiled for a processor class can be inserted instead of multiple copies for specific processors.

When linking a collection of compiled units together into a single linked unit, the user must select a specific processor type or processor class on which the linked unit is to run. As before, this determines the set of processor types on which the code will run. When linking for a particular type or class, the linker will accept compilation units compiled for a compatible class. Table B.2 shows which processor types and classes the linker will accept when linking for a particular class.

For example, if the target processors are an IMS T400 and a T425 the user may compile for classes T5 and TB and link the code for class T5. Code for a different processor class can be included in the final linked unit, as long as:

- It uses the instruction set or a subset, of the instruction set of the link class
- The calling conventions are the same.

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

Table B.2 Linking processor classes

As can be seen from table B.2 certain processor classes are incompatible, e.g. classes T8 and T9 cannot be linked with class TA. The reason why these classes cannot be linked together is explained in section B.2.4. which gives details of the differences between the instruction sets, as additional information.

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

The linker uses the rules given in Table B.2 to determine whether a compiled module, found in a library, is suitable for linking with the current system. So, for example, to create a library which may be linked with any T2/T4/T8-series processor class or specific processor type, all routines could be compiled for classes T2, TA, T8, and processor type ST20 or T450.

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

Note: code compiled for ST20 or T450 processor types cannot be linked with any of the other classes.

occam object files targetted at different targets

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

Table B.2 can be used as guide, by regarding the ‘link class’ as the ‘calling class’ and the ‘processor classes which may be linked’, as the ‘processor classes which may be called’.

Note: classes T8 and T9 cannot call class TA.

Note: At configuration level, code compiled for class TA can be run on a T8 processor, provided the outermost routine does not include any function which returns an arithmetic REAL. This is because of the different methods of evaluating REAL arithmetic for different processor targets, see section B.2.4.

B.2.4 Classes/instruction sets – additional information

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

- Classes T2 and T3 support 16–bit processors whereas all the other processor classes support 32–bit processors.
- Class T3 is the same as class T2 except that T3 has some extra instructions to perform CRC and bit operations, *dup*, double word indexing and includes special debugging functions.
- Class T5 is the same as class T4 except that T5 has extra instructions to perform CRC, 2D block moves, bit operations, double word indexing, special debugging functions and also includes the *dup* instruction.
- Class T9 is the same as class T8 except T9 has additional debugging instructions.
- The T800, T801 and T805 processors use an on-chip floating point processor to perform REAL arithmetic. Thus a large number of floating point instructions are available for these processors and for their associated classes T8 and T9. These instructions are listed in *The Transputer Databook*.
- For the T414, T400, and T425 processors i.e. processor classes T4 and T5 the implementation of REAL arithmetic is in software. These processors make use of a small number of floating point support instructions. Details can be found in *The Transputer Databook*.
- The instruction set of class TA only uses instructions which are common to the T400, T414, T425, T800, T801 and T805 processors. Therefore it does not use the floating point instructions, the floating point support instructions or the extra instructions to perform CRC, 2D block moves or special debugging or bit operations and it does not use the *dup* instruction.
- The instruction set of class TB only uses instructions which are common to the T400, T414, and T425 processors. Therefore it uses the floating point support

instructions, but does not use the extra instructions to perform CRC, 2D block moves or special debugging or bit operations and it does not use the *dup* instruction.

- The runtime model used for T450 and ST20 differs from that of all other processors; for example, the use of hardware semaphores. Therefore T450 and ST20 code cannot be mixed with code for any other processor types. For further details see '*The ST20450 Datasheet - 42-1626-02*'.

When considering the similarities and differences in the instruction sets of different processor classes it helps to divide them into the separate structures as shown in Figure B.1.

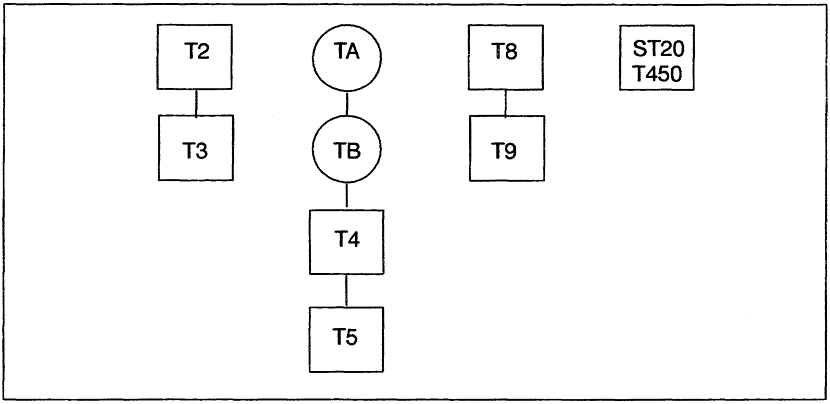


Figure B.1 Structures for mixing processor types and classes

By comparison with Table B.2 it can be seen that a module may only be linked with modules compiled for a processor class which belongs to the same structure.

Classes T2 and T3 are targetted at 16–bit processors so it is obvious that they cannot be linked with the other classes which are all targetted at 32–bit processors.

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

To summarize, compiling code for the processor classes TA and TB enables it to be run on a large number of processor types, however, the code may not be as efficient as code compiled for one of the other processor classes or for a specific processor type. For example compiling code for class T5 enables the CRC and 2D block move instructions to be used, whereas these instructions are not available to code compiled for classes TA and TB.

B.3 Processor type command line options

This section lists the command line options used to specify a target processor or processor class. A target processor must be specified for the following tools:

`icc`	–	The ANSI C compiler.
`oc`	–	The occam 2 compiler.
`ilink`	–	The toolset linker.

Option	Description
`ST20`	Specifies the `ST20` target processor. Same as `T450`.
`TA`	Specifies target processor class TA (T400, T414, T425, T800, T801, T805).
`TB`	Specifies target processor class TB (T400, T414, T425).
`T212`	Specifies an IMS T212 target processor.
`T222`	Specifies an IMS T222 target processor. Same as `T2`.
`M212`	Specifies an IMS M212 target processor. Same as `T2`.
`T2`	Same as `T212`, `T222` and `M212`
`T225`	Specifies an IMS T225 target processor.
`T3`	Same as `T225`.
`T400`	Specifies an IMS T400 target processor. Same as `T425`.
`T414`	Specifies an IMS T414 target processor.
`T4`	Same as `T414`.
`T425`	Specifies an IMS T425 target processor.
`T450`	Specifies an IMS T450 target processor.
`T5`	Same as `T400`, and `T425`.
`T800`	Specifies an IMS T800 target processor.
`T8`	Same as `T800`.
`T801`	Specifies an IMS T801 target processor. Same as `T805`.
`T805`	Specifies an IMS T805 target processor.
`T9`	Same as `T801` and `T805`.

Table B.3 Processor type command line options

C ANSI C compiler optimization examples

C.1 Local optimization examples

This section briefly describes each of the local optimizations supported by the ANSI C optimizing compiler.

C.1.1 Peephole optimization

This optimization is performed by the compiler at assembly code level. The compiler scans the assembly code for sequences of instructions which may be reduced to a faster or more compact sequence of instructions.

Example:

If a source code instruction generated the following assembly code instructions:

```
ldc x
ldc y
and
```

then they could be reduced to the following single instruction:

```
ldc x & y
```

where: the expression `x & y` is evaluated by the compiler (since `x` and `y` are constants).

In a similar manner, the sequence:

```
ldl n
stl n
```

could be removed altogether.

Summary of effects:

- Slight improvement to execution time: some instructions are no longer executed.
- Slight improvement to code size: some instructions are no longer coded.

C.1.2 Flowgraph optimizations

Flowgraph optimizations cover a wide range of local optimizations which are performed on short sequences of code.

The following examples describe typical optimizations of this type.

Branch–chaining optimization

When the destination of one jump is to another jump, then the first jump is replaced with a jump to the destination of the second jump.

This optimization cannot be performed at source code level and is best demonstrated in assembly code:

```
     j  L1
     ...
L1:  j  L2
```

becomes:

```
     j  L2
     ...
L1:  j  L2
```

Dead code elimination

Dead code elimination is the removal of statements which cannot be reached and is another type of flowgraph optimization. For example:

```
void p(void)
{
  while (1)
    ...loop body – contains no break statements
  s;  /* This statement cannot be reached*/
}
```

With dead code elimination, this code segment would be transformed to:

```
void p(void)
{
  while (1)
    ...loop body – contains no break statements
}
```

The statement 's' is deleted.

Summary of effects of flowgraph optimizations:

The effect on both execution time and code size varies on the particular optimization performed. For the examples shown above the results are:

- Branch–chaining – improved execution time and a slight reduction in code size.
- Dead code elimination – code size is improved.

C.1.3 Redundant store elimination

Assignments to variables which are not subsequently used, are deleted by an optimization called redundant store elimination.

Example:

```
void p(void)
{
     int x;
     ... some code
     x = 27;
     ... some more code which does not read x
}
```

With redundant store elimination, this segment of code would be transformed to:

```
void p(void)
{
     int x;
     ... some code
     ... some more code which does not read x
}
```

The assignment of a value to `x` is removed.

Summary of effects:

- Slight improvement to execution time.
- Slight improvement in code size.

C.2 Global optimization examples

This section briefly describes each of the global optimizations supported by the ANSI C optimizing compiler.

C.2.1 Common subexpression elimination

The purpose of common subexpression elimination is to remove from the program any redundant computations. An expression is redundant where it is identical to and computes the same value as another expression whose value is still available for use.

Such commonality is not restricted to explicit computations in the source code but may include implicit computations such as array element address calculation. Subscripted expressions often repeat in blocks of code. Where this happens it is often more efficient to extract expressions which occur more than once so that they are evaluated once only.

Example:

Code segment before common subexpression elimination is applied:

```
{
      int a[10][10], i, j;
      for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
          a[i][j] = a[i][j] + t;
  }
```

Code segment after common subexpression elimination is applied:

```
{
      int a[10][10], i, j;
      for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
          {
            int *temp3 = &a[i][j];
            *temp3 = *temp3 + t;
          }
    }
```

Notice that the subscripted variable in the summation has been replaced by a single variable `*temp3` .

Common subexpression elimination is achieved by saving the result of a computation in a temporary location rather than recomputing the expression.

Summary of effects:

- Improvement to execution time: expressions which were evaluated several times are now only evaluated once.
- Improvement to code size: expressions which were coded several times are now only coded once.

- Increase in workspace size: expressions which were evaluated several times now have their value stored in a temporary variable in workspace.

The compiler evaluates each potential case and only applies the optimization if it is worthwhile.

C.2.2 Loop–invariant code optimization

This optimization removes expressions which remain constant during the execution of a loop, to outside the loop so that they are executed once only. Invariant expressions often include subscripting calculations as well as computations in the source code.

Example:

Code segment before loop–invariant code optimization is applied:

```
{
   int a[10][10], i, j;
   for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
     a[i][j] = a[i][j] + t;
 }
```

Code segment after loop–invariant code optimization is applied:

```
{
   int a[10][10], i, j;
   for (i = 0; i < 10; i++)
    {
     int *temp1 = &a[i];
     int *temp2 = &a[i];
     for (j = 0 ; j < 10; j++)
      temp1[j] = temp2[j] + t
    }
  }
```

In this example the value of `i` remains constant during iterations of the inner loop which increments `j`. The calculation of `&a[i]` can therefore be moved outside the inner loop.

Summary of effects:

- Improvement to execution time: expressions which were evaluated on every loop iteration are now only evaluated once.
- Slight increase in code size: extra code has to be inserted to store the result of an expression in a temporary.
- Increase in workspace size: expressions which were evaluated on every loop iteration are now evaluated into a temporary variable outside of the loop.

C.2.3 Global optimization example

This example is based on the source code used in the previous two sections and shows what happens when both global subexpression elimination and loop–invariant code optimization are applied:

```
  {
       int a[10][10], i, j;
       for (i = 0; i < 10; i++)
         {
           int *temp1 = &a[i];
           for (j = 0; j < 10; j++)
             {
               int *temp3 = temp1[j];
               *temp3 = *temp3 + t;
             }
         }
     }
```

C.2.4 Strength reduction

This optimization replaces an expensive operation by a cheaper one. The compiler examines each loop in the source code, looking for any expression which is a linear function of the number of times the loop is executed. For each such expression found, the compiler introduces a temporary to hold the value of the expression on entry to the loop, and increments the temporary each time round the loop. The evaluation of the expression is replaced by a load from the temporary.

Example:

Code segment before strength reduction optimization is applied:

```
i = 1;
while (i < 10)
  {
    a[i][j] = a[i][j] + b[i][j];
    i += 1;
  }
```

Code segment after strength reduction optimization is applied:

```
{
  int *temp1, *temp2;
  i = 1;
  temp1 = &a[1][j];
  temp2 = &b[1][j];
  while (i < 10)
    {
      *temp1 = *temp1 + *temp2;
      i += 1;
      temp1 += 10;
      temp2 += 10;
    }
}
```

In this example, the addresses of the array accesses, `&a[i][j]` and `&b[i][j]` increase by 10 words on each iteration of the loop. The compiler evaluates the addresses on first entry to the loop, and puts them in temporaries: `temp1` contains the value of `&a[i][j]` on entry to the loop, and `temp2` contains the value of `&b[i][j]` on entry to the loop.

Then on each iteration of the loop, instead of recalculating the addresses, the compiler generates code to load the values of the temporaries. The compiler also inserts code to increment the temporaries each time round the loop.

Summary of effects

Improvement to execution time: expressions which needed recalculation each time round the loop are reduced to a simple addition.

Increase in workspace size: new temporaries are introduced to hold the values of strength–reduced expressions.

Slight increase in code size: extra code has to be inserted to load and increment the temporaries.

C.2.5 Tail–call and tail recursion optimization

The purpose of these optimizations is to make function calls more efficient. When the last operation performed by a function is to call another function, tail–call optimization may be applied. In C programs a function may in fact, call itself, in which case the optimization is called tail recursion optimization.

The optimization is achieved by substituting a jump instruction instead of the call instruction. This optimization cannot be performed at source code level.

When a jump instruction is used, the return from the other function will return the caller to the caller of the current function, thereby saving one return sequence. The called function's workspace is also laid on top of the current function's workspace, thus saving stack size.

Example: (Tail–call optimization)

Take the following code segment:

```
void p(int x)
{
  ...body of p
  q(x+1);
}
```

Without optimization the code generated for routine 'p' would be:

```
p:
            ajw      –3
            ... body of p
            ldl      2          —x
            adc      1
            ldl      1          —<static_link>
            call     $q
            ajw      3
            ret
```

After tail–call optimization, the code generated is:

```
p:
          ajw     –3
          ... body of p
          ldl     2          —x
          adc     1
          stl     2          —x
          ajw     3
          j       $q
```

Note: that the workspace for routine 'q' is overlaid on the workspace for routine 'p'.

Summary of effects: (Tail–call optimization)

- Little effect on execution time.
- Workspace requirements are reduced as the called function's workspace is overlaid on the calling function's workspace.

Example: (Tail–recursion optimization)

```
void p(int x)
{
  ...body of p
  p(x+1);
}
```

This code segment when compiled without optimization would cause the following code to be generated for routine p.

```
p:
          ajw     –3
          ... body of p
          ldl     2          —x
          adc     1
          ldl     1          —<static_link>
          call    $p
          ajw     3
          ret
```

After tail–recursion optimization, the code generated is:

```
p:
          ajw     –3
..3:
          ... body of p
          ldl     2          —x
          adc     1
          stl     2          —x
          j       ..3
```

Note: that the workspace for the second invocation of 'p' is laid on top of the workspace for the first invocation of 'p'. Also note that the second invocation of 'p' does not re–execute the routine entry code (in this example, an 'ajw –3' instruction).

Summary of effects: (Tail–recursion optimization)

- Execution time is improved as the called function's entry sequence is already evaluated. In addition, it may not be necessary to assign the actual parameters to the formal parameters of the function called.

- Workspace requirements are reduced as the called routine's workspace is overlaid on the calling routine's workspace.

C.2.6 Workspace allocation by coloring

This method of workspace allocation can be performed when the lifetimes of two variables, 'a' and 'b' do not overlap. When this is the case 'a' and 'b' may be allocated in the same workspace slot.

For example, in the following segment of code, variables 'a' and 'b' can be placed in the same workspace slot because their values are never required at the same time:

```
{
     int a, b:
     a = funcl(27);
     procl(a);
     procl(a);
     /* 'a' is not used after this point, 'b' is not
    used before this point */
     b = func2(34);
     proc2(b);
   }
```

When optimization is enabled, the compiler will use this method of workspace allocation, provided the code is suitable.

If workspace is allocated by coloring, then the compiler calculates a usage count for each variable, and places the most frequently used variables at lower workspace positions.

When the command line option 'O0' is used i.e. when optimization is disabled, all variables are allocated their own unique workspace slot.

D Using the assembler

This appendix describes the assembler supplied with the ANSI C toolset. The appendix explains how to invoke the assembler and describes the use and syntax of assembler directives. The chapter ends with a list of assembler diagnostic messages which may be obtained.

D.1 Introduction

The assembler is supplied as an integral part of the ANSI C compiler `icc` and is normally run as the final stage of compilation. The command line interface of the compiler enables the user to invoke the assembler directly. This suppresses the compilation phase of the compiler and the input file is passed directly to the assembler.

The assembler is a cross–assembler which enables a source file written in assembly code to be translated into object code. The assembler accepts as input a single ASCII text file consisting of transputer instructions and assembler directives.

If required the compiler preprocessor can be run on an assembler source file and the output file generated by the preprocessor used as input to the assembler.

The assembler generates an object file in *T*ransputer *C*ommon *O*bject *F*ile *F*ormat (TCOFF). This file may then be linked with other TCOFF object files or library modules to produce a linked unit, which may then be configured and/or loaded onto a transputer network and the application executed.

D.2 Running the assembler

The assembler is invoked by using the '`AS`' option of the ANSI C compiler `icc`.

The assembler is invoked to assemble a source file by the following command line:

`icc` *filename.ext* `–as` *{options}*

where: *filename.ext* is the filename and extension of the assembler source file, see section D.2.1.

options is a list of `icc` command line options, see section D.2.2.

A list of error messages which may be generated by the assembler is given in section D.6.

D.2.1 Specifying the source filename

The full filename including extension must always be supplied on the compiler command line when the '`AS`' option is used. If an extension is not supplied the compiler assumes a C source file is to be compiled and searches for an input file which has the appropriate name and '.c' file extension.

D.2.2 Use of icc command options with the assembler

Many of the icc command line options have no meaning if used with the assembler and will be ignored. The only options which have meaning are:

- Any option to select the target processor type (see appendix B).
- 'I' – Displays progress information as the tool runs.
- 'O *filename*' – Specifies an output filename.

icc command line options are documented in full in chapter 1.

D.2.3 Using the preprocessor with the assembler

Preprocessor directives may be included in assembler source files, pragma directives, however, should not.

When preprocessor directives are used, the compiler preprocessor must be run on an assembler source file prior to using the assembler. The output file generated by the preprocessor is then used as input to the assembler. The preprocessor is invoked using the compiler command line option 'PP' and the 'O' option is used to name the output file. This temporary file is then input on the compiler command line and the assembler is run. The 'O' option is used to specify an output file for the assembled code.

Examples:

```
icc test.s –pp –o temp.s
icc temp.s –st20 –as –o test.tco

icc test.s –pp –o temp.s
icc temp.s –t450 –as –o test.tco

icc test.s –pp –o temp.s
icc temp.s –t805 –as –o test.tco
```

D.3 Language

An assembler source file is made up of lines of ASCII text. Each line can contain the following:

- A label definition.
- Assembler commands separated by semi-colons or new lines.
- A comment.

None of the above may extend to more than one line.

An assembler command is one of the following:

- A transputer instruction mnemonic (with its operand, if any)
- An assembler directive.

D.3.1 Label definitions

Label names can contain alphanumeric characters and the characters '.' (full–stop) and '_' (underscore). A label is defined by terminating its name by a colon:

```
label:
```

A label is used by giving its name without the colon, e.g.

```
j label                   -- jump to label
```

Labels are also referred to as code symbols.

D.3.2 Symbols

There are three kinds of symbols in the assembler. They are as follows:

- Code symbols : These are labels as defined above.
- Data symbols : These are symbols introduced by the `data` or `common` directives.
- Defined symbols : These are symbols defined using the `defsym` directive.

Once a symbol has been defined as one kind of symbol it cannot be redefined to another.

D.3.3 Expressions

The assembler can recognize simple integer expressions constructed from operators, operands and parentheses. An operator performs a mathematical or logical operation on the operand(s) in the expression. Allowable operators are listed in table D.1.

Operator	Meaning	Precedence
–	unary minus	1
~	bitwise not	1
!	symbol offset	1
*	multiplication	2
/	division	2
%	modulus	2
+	addition	3
–	subtraction	3
&	bitwise and	4
|	bitwise or	4
^	bitwise exclusive or	4
<	logical left shift	5
>	logical right shift	5

Table D.1 Operators

Evaluation is left-to-right, except for unary operations where the operator closest to the operand binds more tightly, with precedence rules as follows. The lower the number in the precedence column of the table, the higher the precedence of the operator. Parentheses, (...), may be used to alter the order of evaluation of an expression.

All calculations are performed in twos complement arithmetic, modulo 32 bits, with the following exceptions:

- division and modulus by zero will result in a serious error being reported;
- `MOSTNEG_INT / –1` will result in a serious error being reported.

Table D.2 indicates the different types of operand which may appear within expressions:

Operand	Value
Number	Its numerical value.
Defined symbol	The value assigned to the symbol using `defsym`.
Code symbol	The offset in bytes from the end of the instruction containing the expression to the symbol.
Data symbol	The word offset of this symbol in the data area.

Table D.2 Operands

Note: code and data symbols may not appear in the same expression.

When code or data symbols are preceded by the '`!`' operator, the value yielded is the offset of the symbol from the start of the section in which they are defined. This is the offset after the linker has concatenated all of the sections together, and hence this value can only be calculated by the linker.

Mnemonic	Instruction name
adc	add constant
ajw	adjust workspace
call	call
cj	conditional jump
eqc	equals constant
j	jump
ldc	load constant
ldl	load local
ldlp	load local pointer
ldnl	load non-local
ldnlp	load non-local pointer
nfix	negative prefix
opr	operate
pfix	prefix
stl	store local
stnl	store non-local

Table D.3 Transputer primary instructions

D.3.4 Transputer instruction mnemonics

Detailed information on the transputer instruction sets is given in the *Transputer instruction set – a compiler writer's guide* and *The transputer databook* for IMS T2/T4/T8 series transputers. Pseudo instructions are given in the *ANSI C Toolset Language and Libraries Reference Manual.* Primary instructions are listed in Table D.3.

The '*ST20T450 Datasheet - 42-1626-02*' gives details of the ST20 and T450 instruction set.

Note: No check is made that the transputer instructions used in the source code are supported by the transputer target selected on the compiler command line.

D.3.5 Comments

A comment is introduced by the characters ––. If a comment needs to be split over more than one line, each line must start with the characters ––. All text appearing on the same line and to the right of these characters is interpreted as a comment. For example:

```
-- A comment on a line of its own
      j fred  -- A comment at the end of a line
```

D.4 Assembler directives

This section briefly describes each assembler directive and provides an example where appropriate. Directives appear in alphabetical order.

Table D.4 summarizes the directives available.

Directive	Description
`align`	Aligns next generated byte to word boundary.
`blkb`	Generates a block of bytes.
`blkw`	Generates a block of words.
`byte`	Generates a sequence of bytes set to specified values.
`comment`	Causes a comment to be written to the object file.
`common`	Defines a FORTRAN common block.
`data`	Defines a symbol to be a data symbol.
`debug`	Generates a debug information record.
`defsym`	Assigns a value to a symbol. Used only for local symbols within an assembler file.
`descriptor`	Creates an occam style descriptor in the object file.
`extern`	Declares a symbol to be external to the module.
`global`	Defines a symbol to be globally visible.
`init`	Defines a member of the static initialization chain.
`language`	Defines the language of the current module.
`local`	Defines a symbol to be local to the current module.
`maininit`	Used to find the start of the static initialization chain.
`map1`	Generates text information for a memory map file.
`map2`	Generates symbol information for a memory map file.
`map3`	Generates symbol information for a memory map file.
`patch`	Generates a patch. Six patch types are available.
`size`	Pads out an instruction so that it occupies a specified number of bytes.
`sourcefile`	Overrides the default source file name with the specified name.
`textname`	Replaces the default code section name for the current module with the specified name.
`toolname`	Overrides the record of the tool used to create the object file, with the specified tool name.
`word`	Generates a sequence of words set to specified values.

Table D.4 Assembler directives

The names of assembler directives are reserved keywords, as are all transputer instruction mnemonics.

align

Syntax:

```
align
```

Description:

The `align` directive causes the next generated byte in the code section to be aligned on the next word boundary as defined by the target word length of the processor.

blkb

Syntax:

```
blkb <expr> [, <expr_or_string>]
```

Description:

The **blkb** directive generates a block of bytes. The number of bytes to be generated is given by the first expression, the `size`. The value of the bytes is given by the operands which follow the first expression.

If no operands are given then `size` zero bytes are generated.

If the operand is an expression and its value is too large to fit in a byte then an error is reported. See section D.6.

If the operand is a string then the characters of the string are written to consecutive bytes of memory.

If the length of the string exceeds the specified size then the trailing bytes are ignored.

If too few expressions are given then the remaining bytes are set to zero. It is an error to give too many expressions to the **blkb** directive.

Examples:

```
    blkb 10, "hello", 6, 7, 8, 9, 10
```

This generates the byte values 'h', 'e', 'l', 'l', 'o', 6, 7, 8, 9, 10 to consecutive bytes of memory.

blkw

Syntax:

```
blkw <expr> [, <expr>]
```

Description:

The `blkw` directive generates a block of words. The number of words to be generated is given by the first expression, the `size`. The value of the words is given by the operands which follow the first expression. Each of the operands is itself an expression.

If no operands are given then `size` zero words are generated. This result is also obtained if the `size` given is zero or a negative quantity.

If too few operands are given then the remaining words are set to zero. It is an error to give too many operands to the `blkw` directive. Each word is stored in little–endian format.

Examples:

```
    blkw 3 + 4, 1, 2, 3, 4, 5, 6, 7
```

The above generates 7 words in memory with the values 1, 2, 3, 4, 5, 6 and 7.

byte

Syntax:

```
byte <expr_or_string> [, <expr_or_string>]
```

Description:

The byte directive generates a sequence of bytes each of which takes on the values of the following operands.

If the operand is an expression and the result of the expression is too large to fit in a byte then an error is generated. See section D.6.

If the operand is a string then the characters of the string are assigned to consecutive bytes of memory.

Examples:

```
   byte "hello", 6, 7, 8, 9, 10
```

This generates the byte values ’h’, ’e’, ’l’, ’l’, ’o’, 6, 7, 8, 9, 10 to consecutive bytes of memory.

comment

Syntax:

```
comment <string>
```

Description:

This directive causes the string to be written to the object file as a TCOFF comment. The comment is printable but cannot be copied so it will not appear in a linked unit.

A comment can be seen by using the lister tool on an object file. The `ilist` command line option '`m`' is used.

Examples:

To write the comment "Hello" to the object file:

```
comment "Hello"
```

common

Syntax:

```
common <symbol> <expr>
```

Description:

The common directive defines a FORTRAN common block denoted by the symbol symbol, with size in words given by the expression. A common block resides in a TCOFF section of its own.

The final size of a common block is given by the common directive for a given symbol with the greatest size which is present in the link.

This directive is included for completeness, it has no application in C programs.

data

Syntax:

```
data <symbol> <expr>
```

Description:

The `data` directive defines `symbol` as a data symbol. The size in words of the data item is given by the expression and this much space is reserved in the current module's data area. The space is allocated from above the last data symbol or from the start of the area if there were no previous data symbols. The value of a data symbol is its word offset into the current module's data area.

Examples:

To define a data symbol `fred`, 1 word long and to make it visible outside of this module:

```
data fred 1    -- define the data symbol, 1 word long
global fred    -- make it global
```

debug

Description:

The debug directive defines a debug information record and is designed for use by compilers and other programs generating assembly language output. It is listed here for completeness.

defsym

Syntax:

```
defsym <symbol> <expr>
```

Description:

This directive allows a numeric value to be assigned to the symbol `symbol`. The value is given by the expression. This symbol can then be used in expressions as if it were the value itself.

This is only for use internally to an assembler file. These symbols cannot be made global and used in other files. A symbol which is defined using a `defsym` directive must not have been previously defined as a code or data symbol and vice versa.

Code and data symbols may not appear in the expression, only symbols defined using `defsym` are permissible. Any symbol used in the expression must have been previously defined.

Examples:

To set the symbol 'one' to be the value 1:

```
defsym one 1
```

`one` can now be used in expressions as if it were the value 1.

descriptor

Syntax:

`descriptor <symbol> <string> <language_type> <expr> <expr> <string>`

Description:

This directive causes an occam style descriptor to be output to the object file. It also causes two symbols to be defined which are used by some other tools, including the toolset collector, to obtain workspace and vector space information.

The `symbol` is a code symbol denoting the routine for which the descriptor is to be created. This symbol must have previously been declared as global.

The first string is used as a prefix for two symbols which contain the workspace and vector space requirements, (in words) for the routine.

`language_type` is a language type of the same form as that for the `language` directive. The convention is that the language type denotes the source language used in the interface description contained in the descriptor string, however a language type must still be supplied even if the descriptor string is an empty string.

The next two expressions are the workspace and vector space requirements respectively and the last string is the descriptor string itself.

Examples:

To define an occam descriptor for the routine `FRED` which requires 42 words of workspace, no vector space and has the following definition:

```
PROC FRED([18]INT ProcessData)
  SEQ
    ... lots of work
:
```

use the following directive:

```
FRED:
  global FRED
  descriptor fred "FRED" occam 42 0 "PROC FRED([18]INT
ProcessData)\n  SEQ\n:"
```

This generates the following TCOFF records:

```
00000053 SYMBOL EXP UNI "FRED'ws" id: 3
0000005E SYMBOL EXP UNI "FRED'vs" id: 4
00000069 DEFINE_SYMBOL id: 3 42
0000006E DEFINE_SYMBOL id: 4 0
00000073 DESCRIPTOR id: 2 lang: OCCAM
ws: 42 vs: 0
PROC FRED([18]INT ProcessData)
  SEQ
:
```

TCOFF records can be listed with `ilist`, using the command line ‘`t`’ option.

extern

Syntax:

```
extern <symbol> [weak] [realname <string>] [origin <string>]
                [datausage] [routineusage] [notypeusage]
```

Description:

The `extern` directive is used to declare the symbol `symbol` as external to the current module. An example of its use is to call an external routine (see the description of the `patch` directive for an actual example).

The optional `weak` argument indicates that a weak reference is to be made to the symbol. `weak` references do not force the linker to resolve the reference. They are used for example, to resolve debug information, but will not cause the linker to pull in extra library modules simply to resolve debug information.

The name used for the symbol in the object file is the same as the name of the symbol in the assembler source file, unless `realname` is specified. In this case the name used for the symbol in the object file is the name specified by the string argument to `realname`.

`origin` enables an origin to be specified for the symbol in the object file. This allows the linker to ensure that only a definition of a symbol with the same name and the same origin matches this external reference. If `origin` is not specified, then the symbol in the object file is not given a specific origin. The linker can match this external reference with a definition of a symbol with the same name and any origin.

The three optional arguments `datausage`, `routineusage` and `notypeusage` generate extra symbol information in the object file which can be incorporated by the linker into a symbol table. This may be used by tools such as a debugger or profiler, when reading symbol information.

- `datausage` indicates that the symbol is associated with a data object.
- `routineusage` indicates that the symbol is associated with a routine.
- `notypeusage` indicates that the symbol type (data or routine) is not specified.

global

Syntax:

```
global <symbol> [realname <string>] [origin <string>]
                [datausage] [routineusage] [notypeusage]
```

Description:

The `global` directive is used to cause the symbol to become available outside of the current module. It causes the symbol to become globally visible so that it can be accessed by the linker.

The operand to a `global` directive i.e. the symbol, may be a code or data symbol. **Note:** it *must not* be a symbol defined by the `defsym` directive. The `global` directive can appear before the definition of its operand.

The name used for the symbol in the object file is the same as the name of the symbol in the assembler source file, unless `realname` is specified. In this case the name used for the symbol in the object file is the name specified by the string argument to `realname`.

`origin` enables an origin to be specified for the symbol in the object file. This allows the linker to ensure that only a reference to a symbol with the same name and the same origin matches this symbol. If `origin` is not specified, then the symbol in the object file is not given a specific origin. The linker can match this symbol to a symbol with the same name and any origin.

The three optional arguments `datausage`, `routineusage` and `notypeusage` generate extra symbol information in the object file which can be incorporated by the linker into a symbol table. This may be used by tools such as a debugger or profiler, when reading symbol information.

- `datausage` indicates that the symbol is associated with a data object.
- `routineusage` indicates that the symbol is associated with a routine.
- `notypeusage` indicates that the symbol type (data or routine) is not specified.

Examples:

To create a routine called `fred` which can be called from external modules:

```
fred:
        global fred routineusage
        -- do freds operations
```

To create a routine called `fred` which can be called from external modules by the name '`fred's name`', only with the specific origin '`fred's origin`':

```
fred:
        global fred realname "fred's name" origin "fred's origin"
                    routineusage
        -- do freds operations
```

init

Syntax:

`init`

Description:

The `init` directive is used to define a member of the static initialization chain. The static initialization chain is a list of routines which are called by the runtime initialization code, in order to set up the static area.

Each routine in the list is introduced by the `init` directive, which defines the location of a word in memory which is used to link the members of the chain together. After linking, this word holds the byte offset to the next `init` word or zero if it is the end of the chain. The byte directly following the `init` word is the first executable byte of the initialization routine.

Note: that `init` does not itself reserve the word in memory. It is necessary to follow the `init` directive immediately with a directive or dummy instructions to reserve the space. Initialization routines are called with one parameter, a pointer to the start of the static area.

Examples:

For a 32 bit machine:

```
align
init                              -- init directive
byte    #20, #20, #20, #20        -- space reserved for init word
ldc 1                             -- first executable instruction
of                                        -- initialization
routine.
ldc 2
ret                               -- end of initialization routine
```

language

Syntax:

```
language <language_type>
```

Description:

Used to mark a module with its original source language type. For example, the C compiler would mark assembler files that it produced as 'ansi_c'. `language_type` is one of the following:

- `unknown`
- `occam`
- `ansi_c`
- `fortran`
- `iso_pascal`
- `modula2`
- `ada`
- `assembler`
- `occam_harness`

If no `language` directive appears in the input file then the language defaults to `assembler`. The language type is written into the TCOFF `START_MODULE` record.

`occam_harness` is a special language type used to define language runtime system main entry points for use by the configurer.

Examples:

To set the language type for the current file to be ada:

```
language ada
```

local

Syntax:

```
local <symbol> [realname <string>] [origin <string>]
               [datausage] [routineusage] [notypeusage]
```

Description:

The `local` directive is used to declare a symbol which is local to the current module (i.e. not visible outside of the current module). The symbol may be a code or data symbol.

The `local` directive should appear before the definition of its operand.

It is not *necessary* to declare local symbols, as they are automatically declared at their point of definition, however, symbols which are declared with the `local` directive are put into the object file. This means that the `local` directive can be used to ensure that a local symbol's name appears in the object file, and thus be accessible to other tools such as a linker, for insertion in a symbol table.

The name used for the symbol in the object file is the same as the name of the symbol in the assembler source file, unless the `realname` parameter is present. In this case the name used for the symbol in the object file is the name specified by the string argument to `realname`.

If `origin` is specified, then in the object file, the symbol is given the origin specified by the string argument to `origin`.

The three optional arguments `datausage`, `routineusage` and `notypeusage` generate extra symbol information in the object file which can be incorporated by the linker into a symbol table. This may be used by tools such as a debugger or profiler, when reading symbol information.

- `datausage` indicates that the symbol is associated with a data object.
- `routineusage` indicates that the symbol is associated with a routine.
- `notypeusage` indicates that the symbol type (data or routine) is not specified.

Examples:

To create a routine called `fred` which will be named in the object file as `fred`:

```
        local fred routineusage

   fred:
        -- do freds operations
```

To create a routine called `fred` which will be named in the object file as `fred's name` with the specific origin `fred's origin` :

```
        local fred realname "fred's name" origin "fred's origin"
                    routineusage
   fred:
        -- do freds operations
```

maininit

Syntax:

maininit

Description:

The maininit directive is used to find the start of the static initialization chain. The maininit directive defines the location of a word in memory into which the linker will patch the byte offset to the first routine in the static initialization chain. In other words, after linking, the word defined by the maininit patch contains the byte offset to the location of the first init word in the static initialization chain.

Note: that maininit does not itself reserve the word in memory. It is necessary to follow the maininit directive immediately with a directive or dummy instructions to reserve the space.

Note: In order to use a maininit directive, an init directive must be present somewhere in the link. If this is not the case then the link will fail.

Examples:

To obtain the start of the initialization chain on a 32 bit machine:

```
        align
.mainlab:                               -- label so we can find this
                                        -- word
        maininit                        -- maininit directive
        byte    #20, #20, #20, #20      -- space reserved for maininit
                                        -- word
        ldc (.mainlab - .label)         -- load the address of the
                                        -- maininit word
        ldpi                            -- load a pointer to this
                                        -- address
.label:
        ldnl 0                          -- load the contents of the
                                        -- maininit word
                                        -- into the A register
```

map1

map2

map3

Description:

These directives are used internally by the compiler to generate information for a map file. They are listed here for completeness and are not intended for use in customers' assembly source files.

`map1` generates text information, `map2` and `map3` generate symbol information.

patch

Overview:

There are six different types of patch directive. They are:

- CODEFIX
- DATAFIX
- EXTOFFSET
- LIMIT
- MODNUMBER
- STATICFIX

Each is discussed in detail below. **Note:** that no space is reserved by the patch directive. The appropriate number of bytes should be reserved following the patch directive using other directives or dummy instructions.

Each of the six types of patch can come in three forms, they are:

- Instruction. Denoted by the patch directive containing a primary instruction mnemonic. In this case the value of the patch becomes the operand to the instruction and this instruction/operand combination is patched into the hole in the code. If the instruction/operand combination is shorter than the number of bytes reserved in the patch directive then it occupies the start of the reserved space and the unused trailing bytes are filled with pfix 0 instructions (hex 20).
- Short. Denoted by the patch directive containing the word short. In this case the value of the patch is patched directly into the hole in the code reserved for it. The value patched will occupy 2 bytes irrespective of the target processor word length and the hole reserved must reflect this.
- Long. Denoted by the patch directive containing the word long. In this case the value of the patch is patched directly into the hole in the code reserved for it. The value patched will occupy 4 bytes irrespective of the target processor word length and the hole reserved must reflect this.

The first operand of a patch directive is the patch size in bytes. The size of a patch must be between 0 and 255 bytes inclusive. An attempt to issue a patch with a size outside this range will result in an error being reported. Also the size given for a short patch must be two bytes and the size given for a long patch must be four bytes otherwise an error is reported.

If the patch size specified is 0, then the assembler will generate TCOFF directives to instruct the linker to calculate the optimal patch length. In this case, no space should be reserved following the patch directive.

patch – codefix

Syntax:

```
patch <expr> <instruction> codefix <symbol> <expr>
```

Description:

This creates a patch of size n bytes, where n is given by the first expression in the directive. The value of this patch is given by the offset between the byte following the patched instruction or value and the address of the symbol symbol. In the case where the patch is an instruction patch and the patched instruction does not entirely fill the reserved space, the value of the instruction operand is the offset between the first unused trailing byte and the address of the symbol. The symbol must be a code symbol.

The final expression is an offset which is added to the value of the patch. instruction is a transputer primary instruction or the tokens short or long (see the overview of the patch directive for an explanation of these).

Examples:

To call an external function

```
extern  _IMS_printf                   -- declare an external symbol
patch   6 j codefix $_IMS_printf 0    -- do the patch
byte    #20, #20, #20, #20, #20, #20  -- hole for the patch
```

In the above example the patch is six bytes long. The patch produces a j (jump) instruction, the operand of which is the offset between the instruction after the patch and the symbol _IMS_printf. Therefore the jump instruction will transfer control to the instruction at the address of the _IMS_printf symbol. The offset zero in this example, causes a jump directly to the symbol.

To perform the same call, but requesting that the linker calculate the optimal patch size:

```
extern  _IMS_printf                   -- declare an external symbol
patch   0 j codefix $_IMS_printf 0    -- do the patch
                                      -- do not leave a hole
```

patch – datafix

Syntax:

```
patch <expr> <instruction> datafix <symbol> <expr>
```

Description:

This creates a patch of size n bytes, where n is given by the first expression in the directive. The value of this patch is given by the offset, in words, between the start of the local static area for this module and the symbol symbol, plus the value of the second expression, the offset.

The offset can be used to access elements of structures etc. The symbol must be a data symbol. instruction is a transputer primary instruction or the tokens short or long (see the overview of the patch directive for an explanation of these).

Examples:

To obtain the offset of fred from the start of the local static area for this module:

```
   extern fred
   patch 4 long datafix fred 0
   byte #20, #20, #20, #20
```

The above example patches the offset, in words, from the start of the local static area to the location of fred, into the four byte hole.

patch – extoffset

patch <expr> <instruction> extoffset <symbol> <expr>

Description:

This creates a patch of size `n` bytes, where `n` is given by the first expression in the directive. The value of this patch is given by the offset to the symbol `symbol` from the start of the section containing it. If the symbol is a code symbol this offset is in bytes. If the offset is a data symbol the offset is in words. The final expression is an offset which is added to the value of the patch.

`instruction` is a transputer primary instruction or the tokens `short` or `long` (see the overview of the `patch` directive for an explanation of these).

Examples:

To obtain the byte offset to main from the start of the text section:

```
extern main                       -- declare an external symbol
patch 4 long extoffset main 0     -- do the patch
byte #20, #20, #20, #20           -- hole for the patch
```

In the above example the patch is 4 bytes long. The offset of main in the text section is patched into the 4 byte hole.

patch – limit

`patch <expr> <instruction> limit`

Description:

This creates a patch of size `n` bytes, where `n` is given by the first expression in the directive. The value of this patch is given by the size, in words, of the global static area for the entire link. **Note:** that this includes any common blocks which are defined.

`instruction` is a transputer primary instruction or the tokens `short` or `long` (see the overview of the `patch` directive for an explanation of these).

Examples:

To obtain the static size for the program:

```
patch 6 ldc limit
byte #20, #20, #20, #20, #20, #20
```

The above example patches a `ldc` instruction into the 6 byte hole. The operand of the instruction is the size of static used by the program in words.

patch – modnumber

`patch <expr> <instruction> modnumber`

Description:

This creates a patch of size `n` bytes, where `n` is given by the first expression in the directive. The value of this patch is given by the current module number. Each module is identified by a unique module number within a link.

`instruction` is a transputer primary instruction or the tokens `short` or `long` (see the overview of the `patch` directive for an explanation of these).

Examples:

To obtain the module number for the current module:

```
patch 6 ldc modnumber
byte #20, #20, #20, #20, #20, #20
```

The above example patches a `ldc` instruction into the 6 byte hole. The operand of the instruction is the current module number.

patch – staticfix

`patch <expr> <instruction> staticfix [<expr>]`

Description:

This creates a patch of size `n` bytes, where `n` is given by the first expression in the directive. The value of this patch is given by the offset, in words, between the start of the local static area for this module and the global static area for the program, plus the value of the second expression. The second expression is optional; if it is not specified, the value zero is assumed.

`instruction` is a transputer primary instruction or the tokens `short` or `long` (see the overview of the `patch` directive for an explanation of these).

Examples:

To obtain the offset between the local static area for this module and the start of the global static area:

```
   patch 4 long staticfix
   byte #20, #20, #20, #20
```

The above example patches the offset, in words, from the start of the global static area to the start of the local static area, for the current module, into the four byte hole.

size

Syntax:

```
size <expr> <instruction>
```

The `size` directive causes the instruction following to be encoded in exactly `n` bytes, where `n` is given by the expression in the directive. If the instruction can be encoded in less bytes then padding is added after the instruction to increase the size of the instruction to `n` bytes. The padding used is prefix zero instructions.

An error is reported if the size given is too small for the instruction.

Examples:

To force a jump instruction to occupy 4 bytes:

```
   size 4 j label
```

sourcefile

Syntax:

```
sourcefile <string>
```

Description:

The VERSION TCOFF record which appears at the start of all TCOFF object files contains a string which holds the name of the source file used to create the object file. The sourcefile directive causes this name to be changed to the name given by its string operand.

If no sourcefile directive appears in the input then the source file name defaults to the filename given on the command line by the user.

Examples:

To enter the filename fred.s:

```
   sourcefile "fred.s"
```

textname

Syntax:

```
textname <string>
```

Description:

The `textname` directive replaces the default code section name for the current module with the name given in the `string`. This is required in order to perform priority linkage (see chapter 10).

If no `textname` directive appears in the input then the text section name defaults to `text%base`.

Examples:

To change the name of the text section to `fred`:

```
textname "fred"
```

toolname

Syntax:

```
toolname <string>
```

Description:

The VERSION TCOFF record which appears at the start of all TCOFF object files contains a string which holds the name of the tool used to create the object file. The toolname directive causes this name to be changed to the name given by its string operand.

If no toolname directive appears in the input then the tool name string defaults to 'iasm'.

Examples:

To set the tool name to be fred:

```
   toolname "fred"
```

word

Syntax:

```
word <expr> [, <expr>]
```

Description:

The word directive generates a sequence of words containing the values of the expressions. Each word is stored in little–endian format.

Examples:

```
   word 3, 3 + 4
```

This stores the values 3 and 7 to the next two consecutive words in the code section.

D.5 BNF grammar for assembler language

primary-op is a mnemonic, in lower case, of one of the instructions listed in Table D.3 or a pseudo instruction. Pseudo instructions are given in Chapter 4 of the *ANSI C Toolset Language and Libraries Reference Manual. secondary-op* is a mnemonic, in lower case, of any other instruction listed for the appropriate transputer in *Transputer instruction set – a compiler writer's guide* or *The Transputer Databook.* The '*ST20T450 Datasheet - 42-1626-02*' gives details of the ST20 and T450 instruction set.

```
assembler-file   =    line { nl line }

line             =    [ label-def ] [ command-list ] [ comment ]

label-def        =    symbol :

command-list     =    command { separator command }

separator        =    nl
                 |    ;

command          =    tp–instruction
                 |    directive

comment          =    — string

tp-instruction   =    primary-op expression
                 |    secondary-op

primary-op       =    <any primary instruction (in lower case)>

secondary-op     =    <any secondary instruction (in lower case)>

directive        =    align
                 |    blkb expression { , expr-or-string }
                 |    blkw expression { , expression }
                 |    byte expr-or-string { , expr-or-string }
                 |    comment string
                 |    common symbol expression
                 |    data symbol expression
                 |    debug number , number { , number-or-string }
                 |    defsym symbol expression
                 |    descriptor symbol string language-type
                      expression expression string
                 |    extern symbol [weak] [realname string] [origin string]
                                    [datausage] [routineusage]
                                    [notypeusage]
                 |    global symbol [realname string] [origin string]
                                    [datausage] [routineusage]
                                    [notypeusage]
                 |    init
```

```
                      | language language-type
                      | local symbol  [realname string] [origin string]
                                      [datausage] [routineusage]
                                      [notypeusage]
                      | maininit
                      | map1 string
                      | map2 string expression
                      | map3 string expression
                      | patch expression patch-instruction patch-type
                      | size expression tp-instruction
                      | sourcefile string
                      | textname string
                      | toolname string
                      | word expression { , expression }


patch-instruction   =   primary-op
                      | short
                      | long

patch-type          =   codefix symbol expression
                      | datafix symbol expression
                      | staticfix [expression]
                      | modnumber
                      | limit
                      | extoffset symbol expression


language-type       =   unknown
                      | occam
                      | ansi_c
                      | fortran
                      | iso_pascal
                      | modula2
                      | ada
                      | assembler
                      | occam_harness

expr-or-string      =   expression
                      | string

number-or-string    =   number
                      | string

expression          =   number
                      | symbol
                      | monadic-expression
                      | dyadic-expression
                      | ( expression )
```

```
dyadic-expression    =    expression dyadic-operator expression

monadic-expression   =    monadic-operator expression

dyadic-operator      =    *
                        | /
                        | %
                        | +
                        | –
                        | &
                        | |
                        | ^
                        | <
                        | >

monadic-operator     =    -
                        | ~
                        | !

number               =    decimal-number
                        | hex-number

decimal-number       =    <any base 10 number>

hex-number           =    hex-intro <any base 16 number>

hex-intro            =    #
                        | 0x
                        | 0X

symbol               =    <any alphanumeric characters>
                        | .
                        | _  (underscore)

string               =    " <any sequence of printable ASCII characters> "
```

D.6 Errors

There are three levels of errors which can occur in the assembler: *Error, Serious* and *Fatal*. These messages adhere to the standard format for error messages produced by the toolset. This format is documented in Appendix A.

D.6.1 Fatal Errors

These are runtime errors within the assembler. They usually indicate a fault in the assembler. The assembler outputs the error message followed by a banner directing users to seek support. The assembler then exits.

The banner is as follows:

```
***********************************************************
* The assembler has detected an internal inconsistency.   *
* Please contact your supplier who may be able to help    *
* you immediately and will be able to report a suspected  *
* assembler fault to SGS-THOMSON Microelectronics Limited.*
***********************************************************
```

Note: that if the error occurred before the file was opened then the filename and line number are omitted from the error message.

D.6.2 Serious Errors

These are errors from which the assembler cannot recover, e.g. not being able to open a file or out of memory. The assembler will output the error message and then exit.

Note: that if the error occurred before the file was opened then the filename and line number are omitted from the error message.

Arithmetic overflow in expression *operator*

The expression generator has discovered an operation which may result in overflow. The only instance of this at present is the following expression: `MOSTNEG_INT / -1`. Currently *operator* can only be `/`.

Attempt to divide by 0 in expression *operator*

An attempt to divide by zero has been detected in the expression generator. *operator* can be either `/` or `%`.

Cannot open *filename* **for input**

The file *filename* could not be opened for reading.

Cannot open *filename* **for output**

The file *filename* could not be opened for writing.

Cannot open VM file *filename* for output

The assembler's internal virtual memory system has tried to open a file for output and failed.

Error closing object file

A file system error has been detected while closing the object file.

Error writing object file

A file system error has been detected while writing the object file.

Found a map directive with no mapfile

A map directive, `map1`, `map2` or `map3` has been found but no map file is open.

Illegal processor type *string*

The processor type *string* is not recognized as a valid processor type.

Instruction won't fix in <*number*> byte(s) (operand value = <*value*>)

A size directive is too small: it is not possible to encode the number *value* in that number of bytes.

Operator <*operator*> cannot be translated to TCOFF

The TCOFF file format does not support the operator.

Out of memory

The assembler is unable to allocate any more memory.

Token too large: *string*

The token *string* is greater than 1024 characters. Only the first 32 characters of the token are given.

VM file operation failure

An operation on the virtual memory file failed (One of seek, read, write).

D.6.3 Errors

These are errors which the assembler will attempt to recover from. They generally occur while reading the input file. Usually the assembler, on detecting the error, will restart assembly from the next command after the command in which the error occurred. No object file is produced if an error occurs.

***position* operand to *directive–name* must be a numeric expression**

The directive given by *directive–name* has been given an incorrect operand. The operand should be a numeric expression. Which operand is erroneous is given by *position*.

'*symbol–name*' has already been defined with DEFSYM

The symbol *symbol–name* has been defined using `defsym` and has now been discovered in a label definition context. This is illegal.

'*symbol–name*' may not be redefined using DEFSYM

The symbol *symbol–name* has been previously defined somehow. It is now appears in a `defsym` *directive.* This is illegal.

Cannot mix Code and Data symbols in same expression

An expression has been discovered where code and data symbols are mixed. This is illegal.

Colon missing – assumed

The assembler was expecting a colon. It assumes one existed and continues.

Data symbol re–defined as code symbol '*symbol–name*'

The data symbol, *symbol–name,* has been redefined as a code symbol.

Debug: variable dims should be 0

The dimension field of a debug variable record should be zero. A non-zero value has been found.

Duplicate definition of symbol '*symbol–name*'

The symbol *symbol–name* has been previously defined.

Duplicate label definition '*symbol–name*'

The label given by *symbol–name* has been redefined.

Error in data mapping

A data item has been encountered twice at the mapping stage.

Expected number in *directive–name* directive got *symbol*

A number was expected in the directive given by *directive–name,* instead we got the symbol given by *symbol.*

Expected string in *directive–name* directive got *symbol*

A string was expected in the directive given by *directive–name,* instead we got the symbol given by *symbol.*

Illegal *position* operand to *directive–name* directive

The directive given by *directive–name* has been given an incorrect operand. Which operand is erroneous is given by *position.*

Illegal patch size *number*

A patch directive has been encountered with a size operand which is outside the range of legal patch sizes, less than one or greater than 255. The patch size encountered is given by *number.*

Illegal symbol in *directive–name* directive: '*symbol–name*'

The directive given by *directive–name* contains an illegal symbol (given by *symbol–name*).

Instruction won't fit in *number* byte(s)

The size of an instruction requested with the `size` *directive* is too small.

Malformed expression

A badly formed expression has been encountered by the expression parser.

Number too large: <*number*>

number must be representable in 32 bits.

Only numeric expressions or strings allowed in *directive–name*

An operand to the directive, *directive–name*, (one of `byte` or `blkb`) has been found which is not a string or a number.

Operator '!' must have symbol operand, found '<*sting*>'

The `!` operator must have a symbol operand.

Symbol *symbol–name* is undefined

The symbol given by *symbol–name* has been encountered and it is undefined.

Symbol in DESCRIPTOR must be global

The symbol in the `descriptor` *directive* must be declared as global.

Undefined symbol '*symbol–name*' as operand to *directive–name*

The symbol given by *symbol–name* is undefined and has been used in the directive given by *directive–name*. This is illegal.

Unexpected symbol: '*symbol*' *hex-number*

The symbol given by *symbol* and *hex-number* was encountered in the main loop of the assembler parser.

Unexpected symbol '*symbol*' in expression

The symbol *symbol* has been found in an expression by the expression parser. It shouldn't be there.

Value *hex-number* out of range for *directive–name* directive

The value given by *hex-number* has been encountered in the directive given by *directive–name* (one of `byte` or `blkb`) and is not in the range of a byte value.

Wrong length for *patch-type* patch – should be *number*

The length specified for the patch given by *patch-type* (one of `long` or `short`) is incorrect. It should be that given by *number*.

E Memory interface configuration files

This appendix describes the memory interface configurer files, known as memfiles, which are used for initializing the ST20450 memory interface. These files are ASCII text files which may be written or modified by running the Memory Interface Configurer tool, `imem450`, or by using a standard text editor. By convention memfiles have the file extension `.mem`.

E.1 Structure of memfiles

An example memfile is shown in section E.3.

The memfile should have the following structure:

1 Processor type

2 DRAM refresh parameters

3 Global parameters

4 Zero to four memory bank definitions

E.1.1 Processor type

The processor type is specified by the statement `Processor.type`, in Table E.1.

Statement	Meaning	Required
`Processor.type := T450`	Type of processor	Yes

Table E.1 Processor type statement

E.1.2 DRAM refresh parameters definition

The refresh parameters are specified by the statements in Table E.2.

Statement	Meaning	Required
`Dram.refresh.interval :=` *Cycles*	Refresh interval	Optional
`Dram.refresh.time :=` *Cycles*	CAS low time	If interval non-zero
`Dram.refresh.RAS.high :=` *Phases*	RAS high time	

Table E.2 DRAM refresh statements

If `Dram.refresh.interval` is present and non-zero then both the parameters `Dram.refresh.time` and `Dram.refresh.RAS.high` must be present.

E.1.3 Global parameters

The global parameter statements are shown in Table E.3.

Statement	Meaning	Required
`Signal.all.pending.cycles`	Signal any memory cycle	Optional
`Pad.strength.A2.8 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.A9.12 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.A13.16 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.A17.20 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.A21.24 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.A25.31 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.D0.7 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.D8.15 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.D16.31 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.be1 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.be2 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.rcp0 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.rcp1 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.rcp2 :=` *Strength*	Pad driving strength	Yes
`Pad.strength.rcp3 :=` *Strength*	Pad driving strength	Yes
`Proc.clock.out := enabled | disabled`	**ProcClockOut** pin enabled	Yes

Table E.3 Global parameter statements

E.1.4 Bank definitions

There can be up to four memory bank definitions in the memory configuration file for up to four of the memory banks of the IMS T9000. Each bank definition has the following structure:

1 Bank header

2 Memory type

3 Bank description

4 Bank terminator

In Tables E.4 and E.5, the 'Required' column shows whether the statement is required in every bank definition.

The first statement of each bank is the bank header which introduces a bank definition. The last statement of each bank is the bank terminator.

Statement	Meaning	Required
`Bank` *BankNumber* [*"Title"*]	Start of bank definition	Yes
`End.bank`	End of bank definition	Yes

Table E.4 Bank header and terminator statements

The type of the memory in the bank is specified as DRAM or non-DRAM. This defines whether refresh is provided and which statements are expected in the following bank definition. The statement must be the first statement of the bank definition after the `Bank` statement.

Statement	Meaning	Required
`Device.type := dram | non–dram`	DRAM or non-DRAM	Yes

Table E.5 Memory type statement

A DRAM bank definition may contain any of the statements in Table E.6. These statements may appear in any order after the bank header and `Device.type := dram` statements and before the bank terminator. In Table E.6, the 'Required' column shows whether the statement is required in every DRAM bank definition.

Statement	Meaning	Required
`Page.address.bits :=` *HexMask*	Mask page address	Yes
`Page.address.shift :=` *BitShift* `[bits]`	Shift of page address	Optional
`Disable.page.mode`	Disable page mode	Optional
`Port.size :=` *BitWidth* `[bits]`	Width of memory	Optional
`Disable.refresh`	Do not refresh bank	Optional
`Wait.pin := enabled | disabled`	Whether wait pin enabled	Yes
`Ras.cycle.time :=` *Cycles*	RAS cycle time	Yes
`Cas.cycle.time :=` *Cycles*	CAS cycle time	Yes
`Data.drive.delay :=` *Phases*	Data bus write drive delay	Optional
`Ras.edge.time :=` *Phases*	Falling edge delay	Yes
`Ras.edge.active.during :=` *CycleType*	Cycles when RAS edge active	Optional
`Ras.precharge.time :=` *Cycles*	RAS pre-charge time	Yes
`Bus.release.time :=` *Cycles*	Bus release time	Yes

Table E.6 DRAM bank description statements

The definition of a DRAM bank should normally include timing definitions of one or more of the following strobes, as described in Section E.1.5:–

- Programmable strobe (**notMemPS**)
- RAS strobe (**notMemRAS**)
- CAS strobe (**notMemCAS**)
- Write strobe (**notMemWrB0-3**)

A non-DRAM bank definition may contain any of the statements in Table E.7. These statements may appear in any order after the bank header and `Device.type := non–dram` statements and before the bank terminator. In Table E.7, the 'Required' column shows whether the statement is required in every non-DRAM bank definition.

Statement	Meaning	Required
`Port.size :=` *BitsWidth* `[bits]`	Width of memory	Optional
`Wait.pin := enabled` | `disabled`	Whether wait pin enabled	Optional
`Cas.cycle.time :=` *Cycles*	CAS cycle time	Yes
`Data.drive.delay :=` *Phases*	Data bus write drive delay	Optional
`Bus.release.time :=` *Cycles*	Bus release time	Yes

Table E.7 non-DRAM bank description statements

The definition of a non-DRAM bank may include definitions of any of the following strobes, as described in Section E.1.5:–

- Programmable strobe (**notMemPS**)
- RAS strobe (**notMemRAS**)
- CAS strobe (**notMemCAS**)
- Write strobe (**notMemWrB0-3**)

E.1.5 Strobe definitions

A strobe definition is structured in the following way :–

1 Strobe definition header

2 Strobe description statements

3 Strobe definition terminator

In Tables E.8, E.9 and E.10, the 'Required' column shows whether the statement is required in every strobe definition.

A strobe definition is prefaced by one of the strobe definition header statements, listed in Table E.8. The choice of header defines the processor pin which will produce the signals. The optional quoted title string is used by the `imem` program to label timing diagrams.

Statement	Meaning	Required
`Programmable.strobe` ["*Title*"]	Start programmable strobe	One of these
`RAS.strobe` ["*Title*"]	Start RAS strobe definition	
`CAS.strobe` ["*Title*"]	Start CAS strobe definition	
`Write.strobe` ["*Title*"]	Start write strobe definition	

Table E.8 Strobe definition headers

A strobe definition is terminated by the statement `End.strobe`. It has no parameters. This statement must be present after each strobe definition and before the next.

Statement	Meaning	Required
`End.strobe`	End strobe definition	Yes

Table E.9 Strobe definition terminator

The statements used to describe a strobe are listed in Table E.10. The `Inactive` statement may not appear in the same strobe definition as any of the other statements listed.

Statement	Meaning	Required
`Inactive`	Strobe not active	Optional
`Falling.edge.active.during :=` *CycleType*	Read and/or write active	Optional
`Rising.edge.active.during :=` *CycleType*	Read and/or write active	Optional
`Time.to.falling.edge :=` *Phases*	Delay of strobe	Usually
`Time.to.rising.edge :=` *Phases*	Time to end of strobe	Optional

Table E.10 Strobe description statements

E.2 Memfile statements

A memfile consists of a sequence of statements and comments. A comment is a blank line or any text following the comment marker '--' up to the end of the line. Comments are ignored by all tools.

All other lines within the file are statements. Statements and parameters can be provided in either upper or lower case — these are treated as equivalent, except in strings inside quotation marks ("). Some of the statements are assignments of the form *keyword* := *value*. Other statements may have one or more parameters separated from the initial keyword and from each other by spaces or tabs. The statements are defined in section E.2.2 and a summary is given in Table E.11.

E.2.1 Timing parameters

In Table E.11 parameters named *Cycles* or *Phases* are timing parameters. All timing parameters may be given in either processor clock *cycles* or *phases*, with the exception of the DRAM refresh interval, which *must* be specified in processor clock cycles.

For a processor of speed m MHz, one cycle is $\frac{1}{m}$ microseconds. A phase is half a cycle. If the processor speed is m MHz then:

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

The units are specified by using one of the keywords `phase`, `phases`, `cycle`, or `cycles` after the value. Parameters which are shown as *Cycles* may be specified in phases, but the value given must be a whole multiple of two (i.e. a whole number of cycles) and the number of cycles must be in the range shown in Table E.11. Similarly, parameters which are shown as *Phases* may be specified in cycles; the equivalent number of phases must be in the range shown in Table E.11.

E.2 Memfile statements

Statement	Parameters	Values
Bank	*BankNumber* [*"Title"*]	0 .. 3 string
Bus.release.time :=	*Cycles*	0 .. 15 cycle[s]
Cas.cycle.time :=	*Cycles*	2 .. 15 cycle[s]
Cas.strobe	[*"Title"*]	string
Data.drive.delay :=	*Phases*	0 .. 3
Device.type :=	*DeviceType*	dram | non–dram
Disable.page.mode	none	
Disable.refresh	none	
Dram.refresh.interval :=	*Cycles* [cycles]	0 .. 4095
Dram.refresh.ras.high :=	*Phases*	0 .. 31 phase[s]
Dram.refresh.time :=	*Cycles*	1 .. 15 cycle[s]
End.bank	none	
End.strobe	none	
Falling.edge.active.during :=	*CycleType*	read | write | read & write
Inactive	none	
Pad.strength.A2.8 :=	*Strength*	0 .. 3
Pad.strength.A9.12 :=	*Strength*	0 .. 3
Pad.strength.A13.16 :=	*Strength*	0 .. 3
Pad.strength.A17.20 :=	*Strength*	0 .. 3
Pad.strength.A21.24 :=	*Strength*	0 .. 3
Pad.strength.A25.31 :=	*Strength*	0 .. 3
Pad.strength.be1 :=	*Strength*	0 .. 3
Pad.strength.be2 :=	*Strength*	0 .. 3
Pad.strength.D0.7 :=	*Strength*	0 .. 3
Pad.strength.D8.15 :=	*Strength*	0 .. 3
Pad.strength.D16.31 :=	*Strength*	0 .. 3
Pad.strength.rcp0 :=	*Strength*	0 .. 3
Pad.strength.rcp1 :=	*Strength*	0 .. 3
Pad.strength.rcp2 :=	*Strength*	0 .. 3
Pad.strength.rcp3 :=	*Strength*	0 .. 3
Page.address.bits :=	*HexMask*	00000000 .. FFFFFFFC
Page.address.shift :=	*BitShift* [bits]	0 .. 31
Port.size :=	*BitsWidth* [bits]	8 | 16 | 32
Proc.clock.out :=	*WhetherEnabled*	enabled | disabled
Processor.type :=	*ProcessorType*	T450
Programmable.strobe	[*"Title"*]	string
Ras.cycle.time :=	*Cycles*	1 .. 3 cycle[s]
Ras.edge.active.during :=	*CycleType*	read | write | read & write
Ras.edge.time :=	*Phases*	0 .. 7 phase[s]
Ras.precharge.time :=	Cycles	0 .. 15 cycle[s]
Ras.strobe	[*"Title"*]	string

`Rising.edge.active.during :=`	*CycleType*	`read` I `write` I `read & write`
`Signal.all.pending.cycles`	none	
`Time.to.falling.edge :=`	*Phases*	`0 .. 31 phase[s]`
`Time.to.rising.edge :=`	*Phases*	`0 .. 31 phase[s]`
`Wait.pin :=`	*WhetherEnabled*	`enabled` I `disabled`
`Write.strobe`	[*"Title"*]	string

Table E.11 Memfile statement parameters

E.2.2 Statement definitions

`Bank` *BankNumber* [*" Title"*]

This statement starts a memory bank definition. The parameter *BankNumber* selects one of the four banks and must be in the range 0 to 3. The optional *Title* string is used by the memory interface program to label timing diagrams. One `Bank` statement must appear as the first line of each bank definition.

`Bus.release.time :=` *Cycles* `cycle[s]` I `phase[s]`

This statement defines the bus release time for the devices in the current bank. If the keyword `cycles` (or `cycle`) is used then the value of *Cycles* must be in the range 0 to 15 and will be written in the **BusReleaseTime** field of the **ConfigDataField1** register for the current bank. If the keyword `phases` (or `phase`) is used then the value of *Cycles* must be in the range 0 to 30 and must be a whole number of cycles, i.e. divisible by 2, as the value *Cycles* will be divided by 2 and the result written in the **BusReleaseTime** field.

This statement must be present in each bank definition.

`Cas.cycle.time :=` *Cycles* `cycle[s]` I `phase[s]`

This statement defines the CAS cycle time in the current bank. If the keyword `cycles` (or `cycle`) is used then the value of *Cycles* must be in the range 2 to 15 and will be written in the **CAStime** field of the **ConfigDataField1** register for the current bank. If the keyword `phases` (or `phase`) is used then the value of *Cycles* must be in the range 4 to 30 and must be a whole number of cycles, i.e. divisible by 2, as the value *Cycles* will be divided by 2 and the result written in the **CAStime** field.

This statement must be present in each bank definition.

`Cas.strobe` [*"Title"*]

This statement begins the definition of CAS strobe for the current dram bank, which is output on the **notMemCAS** pin. The optional *Title* string is used by `imem` in output displays.

This statement is optional and may appear in any dram bank definition. If a dram bank definition does not include this statement then the CAS strobe for that bank will be assumed to be inactive.

`Data.drive.delay := ` *Phases* `phase[s] | cycle[s]`

This statement defines the drive delay of the data bus at the first of one or more write cycles for the current bank. The delay is the time from the start of **CASTime** before the data pins are driven.

If the keyword `phase` or `phases` is used then the value of *Phases* must be in the range 0 to 3 and it will be written into the **DataDriveDelay** field of the **ConfigDataField2** register. If the keyword `cycle` or `cycles` is used then the value of *Phases* must be in the range 0 to 1 and it will multiplied by 2 and the result written into the **DataDriveDelay** field.

This statement is optional and may be included in the bank specification. If it is not present then the default value of zero is used.

`Device.type dram | non–dram`

This statement specifies the type of device which will be accessed by the current bank, specifying whether or not the bank is attached devices which need refresh. The legal values are `dram` and `non–dram`. Using `non-dram` disables refresh for the current bank.

This statement must be present as the first statement in each bank definition after the `Bank` statement.

Note that using `dram` also causes the tools reading the file to assume multiplexed address pins, using the RAS and CAS strobes. This means that the `Page.address.bits`, `Page.address.shift`, `Ras.cycle.time` and `Ras.precharge.time` statements are expected. For non–multiplexed dram all these values should be set to zero.

`Disable.page.mode`

The presence of this statement signifies that page mode will be disabled for the current bank. This statement may only appear in a DRAM bank definition and is optional. If it is not present in a DRAM bank definition then page mode may be used when two or more successive memory accesses are made to the same memory page (i.e. the same row). The **PageMode** bit in the **ConfigDataField0** register will be set for DRAM banks unless this statement is present for the bank.

`Disable.refresh`

This statement signifies that the devices in this bank are not to be refreshed. If this statement is present in the definition of a bank *n* then the **DRAM***n* bit of the **ConfigDataField1** register will be set to 0; otherwise the default value of 1 will be written for DRAM or 0 for non-DRAM. Setting the **DRAM***n* to 1 enables refresh for bank *n* if `Dram.refresh.interval` is non-zero. This statement is optional. It may only appear in a bank definition for a DRAM type bank.

`Dram.refresh.interval := ` *Cycles* `[cycle[s]]`

This statement specifies the time between successive refresh cycles in cycles. The value *Cycles* will be written in the **RefreshInterval** field of the

ConfigDataField1 register. *Cycles* must be a value in the range 0..4095, optionally followed by the word `cycles`. If a non-zero value is given, then all DRAM banks which include the statement `Device.type dram` will be refreshed.

This statement is optional and may be included in the refresh specification. If it is not present then the default value of zero is used.

A zero value for the refresh interval means there is no refresh, and the following statements must not then be present in the memory configuration file:

`Dram.refresh.time`

`Dram.refresh.ras.high`

`Dram.refresh.ras.high := ` *Phases* `phase[s] | cycle[s]`

This statement defines the time for which the RAS strobe remains high, timing from the start of the refresh time. This statement must be present if the refresh interval is non-zero.

If the keyword `phase` or `phases` is used then the value of *Phases* must be in the range 0 to 31 and it will be written into the **RefreshRASedgeTime** field of the **ConfigDataField1** register. If the keyword `cycle` or `cycles` is used then the value of *Phases* must be in the range 0 to 15 and it will multiplied by 2 and the result written into the **RefreshRASedgeTime** field.

`Dram.refresh.time := ` *Cycles* `cycle[s] | phase[s]`

This statement gives a refresh time for the current bank as a number of cycles, in the range 1..15 cycles. The refresh time for the current bank is the time between CAS going low at the start of the refresh cycle and CAS going high at the end of the refresh cycle.

If the keyword `cycles` (or `cycle`) is used then the value of *Cycles* must be in the range 1 to 15 and it will be written in the **RefreshTime** field of the appropriate configuration register. If the keyword `phases` (or `phase`) is used then the value of *Cycles* must be in the range 2 to 30 and must be a whole number of cycles, i.e. divisible by 2, as the value will be divided by 2 and the result written in the **RefreshTime** field.

This statement must be present in the refresh specification if the refresh interval is non-zero.

`End.bank`

This statement ends a memory bank definition. This statement must be present as the last line of each memory bank definition.

`End.strobe`

This statement ends a strobe definition. This statement must be present as the last statement in each strobe definition. Strobe definitions may only appear in bank definitions.

`Falling.edge.active.during := read|write|read & write|write & read`

This statement specifies in which type of cycle (i.e. read cycles, write cycles or both) the current strobe will have a falling edge during CAS time. This does not affect the RAS strobe falling edge during RAS time. The value assigned to `Falling.edge.active.during` determines the value written to the appropriate `xxxlactive` field of the **ConfigDataField2** register.

This statement is optional. It must not be present in the definition of an inactive strobe. If not specified, the current strobe will go low in both read and write cycles, unless the current strobe is the Write Strobe, in which case it will go low in write cycles only.

`Inactive`

This statement is used to explicitly define the current strobe as being inactive. This statement is optional and has the same effect as omitting the strobe definition from the current bank. If it is present there should be no other statements in the body of the strobe definition.

`Pad.strength.A2.8 :=` *Strength*

This statement is used to define the drive strength of the external address bus pads **MemAddr2-8, notMemBE0** and **notMemBE3**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **A2-8** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.A9.12 :=` *Strength*

This statement is used to define the drive strength of the external address bus pads **MemAddr9-12**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **A9-12** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.A13.16 :=` *Strength*

This statement is used to define the drive strength of the external address bus pads **MemAddr13-16**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **A13-16** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.A17.20 :=` *Strength*

This statement is used to define the drive strength of the external address bus pads **MemAddr17-20**. The value of *Strength* must be in the range 0 to 3, where

0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **A17-20** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.A21.24 :=` *Strength*

This statement is used to define the drive strength of the external address bus pads **MemAddr21-24**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **A21-24** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.A25.31 :=` *Strength*

This statement is used to define the drive strength of the external address bus pads **MemAddr25-31**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **A25-31** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.be1 :=` *Strength*

This statement is used to define the drive strength of the external byte enable strobe pad **notMemBE1**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **BE1** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.be2 :=` *Strength*

This statement is used to define the drive strength of the external byte enable strobe pad **notMemBE2**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **BE2** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.D0.7 :=` *Strength*

This statement is used to define the drive strength of the external data bus pads **MemData0-7**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **D0-7** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.D8.15 :=` *Strength*

This statement is used to define the drive strength of the external data bus pads **MemData8-15**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **D8-15** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.D16.31 :=` *Strength*

This statement is used to define the drive strength of the external data bus pads **MemData16-31**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **D16-31** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.rcp0 :=` *Strength*

This statement is used to define the drive strength of the external strobe pads **notMemRAS0**, **notMemCAS0** and **notMemPS0**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **RCP0** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.rcp1 :=` *Strength*

This statement is used to define the drive strength of the external strobe pads **notMemRAS1**, **notMemCAS1** and **notMemPS1**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **RCP1** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.rcp2 :=` *Strength*

This statement is used to define the drive strength of the external strobe pads **notMemRAS2**, **notMemCAS2** and **notMemPS2**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **RCP2** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Pad.strength.rcp3 :=` *Strength*

This statement is used to define the drive strength of the external strobe pads **notMemRAS3**, **notMemCAS3** and **notMemPS3**. The value of *Strength* must be in the range 0 to 3, where 0 signifies the weakest drive and 3 the strongest. The value of *Strength* will be written into the **RCP3** field of the **PadDrive** register.

This statement must be present in the memory configuration.

`Page.address.bits :=` *HexMask*

This statement defines the mask to select the page address bits to be output on the address bus during a RAS cycle in the current bank. *HexMask* is in hexadecimal, using no prefix. Bits 0 and 1 must be clear. The most significant 30 bits of the value *HexMask* is placed in the **RASbits** field of the **ConfigDataField0** register for the current bank.

This statement must be present in each dram bank definition.

`Page.address.shift :=` *BitsShift* `[bits]`

This statement controls the right shift of the page address prior to outputting the page address on the address bus during the RAS part of the cycle in the current bank. Its purpose is to enable multiplexing of the address for memory devices with multiplexed address pins. Legal values of *BitsShift* are in the range 0 to 31 and are checked with the page address bits mask to ensure no bits are lost. The value may optionally be followed by the word `bits`. The value *BitsShift* is inserted in the **ShiftAmount** field of the appropriate configuration register for the current bank.

This statement is optional and can only appear in a dram bank definition. If unspecified then the shift will be set such that a page address of *m* bits appears on the least significant address pins (excluding the byte select pins), which are:

- pins 0 to *m*–1 for an 8-bit port size
- pins 1 to *m* for a 16-bit port size
- pins 2 to *m*+1 for a 32-bit port size

`Port.size :=` *BitWidth* `[bits]`

This statement defines the width in bits of the devices in the current bank. Legal values of *BitWidth* are 32, 16 and 8. The value may optionally be followed by the word `bits`. The value of *BitWidth* determines the value written to the **PortSize** field of the appropriate configuration register for the current bank.

This statement is optional. It may only appear in a bank definition. If this statement is not present then the default value of 32 bits wide will be used.

`Proc.clock.out := enabled | disabled`

This statement defines whether an output signal will appear on the **ProcClockOut** pin. If `Proc.clock.out` is assigned the value `enabled` then the **ProcClockEnable** bit in the **PadDrive** register will be set.

This statement must be present in the memory configuration.

`Processor.type := T450`

This statement defines the processor type. `T450` is the only legal value. This statement must be the first statement in the file.

`Programmable.strobe` [*"Title"*]

This statement begins the definition of the programmable strobe for the current bank, which is output on the **notMemPS** pin. The optional *Title* string is used by `imem450` in output displays.

This statement is optional. If a bank definition does not include this statement then **notMemPS** for that bank will be assumed to be inactive.

`Ras.cycle.time := `*Cycles* `cycle[s] | phase[s]`

This statement defines the RAS sub-cycle time for DRAM in the current bank. If the keyword `cycles` (or `cycle`) is used then the value of *Cycles* must be in the range 1 to 15 and will be written in the **RASTime** field of the appropriate configuration register for the current bank. If the keyword `phases` (or `phase`) is used then the value of *Cycles* must be in the range 2 to 30 and must be a whole number of cycles, i.e. divisible by 2, as the value *Cycles* will be divided by 2 and the result written in the **RASTime** field for the current bank.

This statement must be present in each dram bank definition.

`Ras.edge.active.during := read | write | read & write | write & read`

This statement specifies in which type of cycle the RAS strobe will become active during RAS time, i.e. read cycles, write cycles or both. The value assigned to `Ras.edge.active.during` determines the value written to the appropriate `RASedgeActive` field of the **ConfigDataField2** register.

This statement is optional. It must not be present in the definition of an inactive strobe. If not specified, the RAS strobe will be active in both read and write cycles.

`Ras.edge.time := `*Phases* `phase[s] | cycle[s]`

This statement defines the delay from the start of the RAS cycle until the RAS strobe goes low for dram in the current bank. If the keyword `phases` (or `phase`) is used then the value of *Phases* must be in the range 0 to 63 and will be written in the **RASEdgeTime** field of the appropriate configuration register for the current bank. If the keyword `cycles` (or `cycle`) is used then the value of *Phases* must be in the range and the result written in the **RASEdgeTime** field of the **TimingControl** register for the current bank.

This statement must be present in each dram bank definition.

`Ras.precharge.time := `*Cycles* `cycle[s] | phase[s]`

This statement defines the RAS pre-charge time for dram in the current bank. If the keyword `cycles` (or `cycle`) is used then the value of *Cycles* must be in the range 0 to 15 and will be written in the **PrechargeTime** field of the **TimingControl** register for the current bank. If the keyword `phases` (or `phase`) is used then the value of *Cycles* must be in the range 0 to 30 and must be a whole number of cycles, i.e. divisible by 2, as the value *Cycles* will be divided by 2 and the result written in the **PrechargeTime** field of the **TimingControl** register for the current bank.

This statement must be present in each dram bank definition.

`Ras.strobe` ["*Title*"]

This statement begins the definition of RAS strobe for the current dram bank, which is output on the **notMemRAS** pin. The optional *Title* string is used by `imem` in output displays.

This statement is optional and may appear in any dram bank definition. If a dram bank definition does not include this statement then the RAS strobe for that bank will be assumed to be inactive.

`Rising.edge.active.during := read | write | read & write | write & read`

This statement specifies in which type of cycle the current strobe will rise before the end of the cycle, i.e. read cycles, write cycles or both. The value assigned to `Rising.edge.active.during` determines the value written to the appropriate xxx`2active` field of the **ConfigDataField2** register.

This statement is optional. It must not be present in the definition of an inactive strobe. If not specified, the rising edge of the current strobe will occur in both read and write cycles, unless the current strobe is the Write Strobe, in which case it will occur in write cycles only.

`Signal.all.pending.cycles`

This statement specifies that the **MemRefreshPending** pin is used to indicate that a memory cycle is pending during DMA; otherwise the pin indicates that a refresh is pending. If this statement is present then 1 will be written to the **CyclePendingMask** bit of the **ConfigDataField1** register; otherwise a 0 will be written. This statement is optional.

`Time.to.falling.edge :=` *Phases* `phase[s] | cycle[s]`

This statement specifies the time delay until the current strobe goes low, timed from the start of the CAS time for DRAMs or the start of the cycle for non-DRAMs. If the keyword `phases` (or `phase`) is used then the value of *Phases* must be in the range 0 to 63 and will be written in the **E1Time** field of the strobe timing register for the current strobe. If the keyword `cycles` (or `cycle`) is used then the value of *Phases* must be in the range and the result written in the **E1Time** field of the strobe timing register for the current strobe.

This statement should always be present in an active strobe definition, except that it may be omitted in the case of the RAS strobe in a DRAM bank.

`Time.to.rising.edge :=` *Phases* `phase[s] | cycle[s]`

This statement specifies the time delay until the current strobe goes low, timed from the start of the CAS time for DRAMs or the start of the cycle for non-DRAMs. If the keyword `phases` (or `phase`) is used then the value of *Phases* must be in the range 0 to 63 and will be written in the **E2Time** field of the strobe timing register for the current strobe. If the keyword `cycles` (or `cycle`) is used then the value of *Phases* must be in the range and the result written in the **E2Time** field of the strobe timing register for the current strobe.

This statement is optional and may appear in any strobe definition. If it is omitted then the strobe will stay low, after `Time.to.falling.edge`, until the end of the cycle.

`Wait.pin := enabled | disabled`

This statement controls whether or not the wait pin, **MemWait**, affects strobe timing of the current bank. If `Wait.pin` is assigned the value `enabled` then 1 will be written to the **MemWaitEnable** bit of the **ConfigDataField1** register of the current bank; otherwise a zero will be written.

`Write.strobe` ["*Title*"]

This statement begins the definition of the write strobe timing for the current bank, which is controlled by the the register **WriteStrobe**. This controls the behavior of all four **notMemBE0-3** pins when addressing the current bank. The optional *Title* string is used by `imem` in output displays.

This statement is optional. If a bank definition does not include this statement then the write strobes will be assumed to be inactive for that bank.

E.3 Example file

```
-- -----
--
--      Memory configuration file produced
--      by the MEM package.
--      on Thu Mar  2 12:55:30 1995
--
-- -----

Processor.Type         := T450
Dram.Refresh.Interval := 320 Cycles
Dram.Refresh.Time      := 4 Cycles
Dram.Refresh.RAS.High := 3 Phases
Signal.All.Pending.Cycles
Proc.Clock.Out        := enabled

Pad.Strength.Rcp0     := 2
Pad.Strength.Rcp1     := 2
Pad.Strength.Rcp2     := 2
Pad.Strength.Rcp3     := 2
Pad.Strength.Be1      := 2
Pad.Strength.Be2      := 2
Pad.Strength.A2.8     := 2
Pad.Strength.A9.12    := 2
Pad.Strength.A13.16   := 2
Pad.Strength.A17.20   := 2
Pad.Strength.A21.24   := 2
Pad.Strength.A25.31   := 2
Pad.Strength.D0.7     := 2
Pad.Strength.D8.15    := 2
Pad.Strength.D16.31   := 2

Bank 0 "SRAM"
    Device.Type                := Non-Dram
    Wait.Pin                   := disabled
    Port.Size                  := 32 bits

    Ras.Strobe "S0"
        Inactive
    End.Strobe

    Cas.Strobe "CAS"
        Falling.Edge.Active.During        := Read & Write
        Rising.Edge.Active.During         := Read & Write
        Time.To.Falling.Edge              := 0 Phases
        Time.To.Rising.Edge               := 4 Phases
    End.Strobe

    Programmable.Strobe "Read strobe"
        Falling.Edge.Active.During        := Read
        Rising.Edge.Active.During         := Read
        Time.To.Falling.Edge              := 0 Phases
        Time.To.Rising.Edge               := 4 Phases
    End.Strobe

    Write.Strobe "Write strobe"
        Falling.Edge.Active.During        := Write
        Rising.Edge.Active.During         := Write
        Time.To.Falling.Edge              := 1 Phases
        Time.To.Rising.Edge               := 3 Phases
    End.Strobe

    Cas.Cycle.Time             := 2 Cycles
    Bus.Release.Time           := 1 Cycles
    Data.Drive.Delay           := 0 Phases

End.Bank

Bank 1 "DRAM"
    Device.Type                := Dram
    Wait.Pin                   := disabled
    Port.Size                  := 32 bits
    Page.Address.Bits          := 003FF000
    Page.Address.Shift         := 10 bits

    Ras.Strobe "RAS"
        Falling.Edge.Active.During        := Read & Write
        Time.To.Falling.Edge              := 0 Phases
    End.Strobe

    Cas.Strobe "CAS"
        Falling.Edge.Active.During        := Read & Write
        Rising.Edge.Active.During         := Read & Write
        Time.To.Falling.Edge              := 2 Phases
        Time.To.Rising.Edge               := 12 Phases
        End.Strobe

        Programmable.Strobe "Write"
            Falling.Edge.Active.During       := Write
            Rising.Edge.Active.During        := Write
            Time.To.Falling.Edge             := 0 Phases
            Time.To.Rising.Edge              := 12 Phases
        End.Strobe

        Write.Strobe "CE"
            Falling.Edge.Active.During       := Read & Write
            Rising.Edge.Active.During        := Read & Write
            Time.To.Falling.Edge             := 0 Phases
            Time.To.Rising.Edge              := 12 Phases
        End.Strobe

        Ras.Precharge.Time       := 2 Cycles
        Ras.Edge.Time            := 2 Phases
        Ras.Edge.Active.During   := Read & Write
        Ras.Cycle.Time           := 2 Cycles
        Cas.Cycle.Time           := 6 Cycles
        Bus.Release.Time         := 1 Cycles
        Data.Drive.Delay         := 0 Phases

    End.Bank
```

F ANSI C configuration language

This appendix describes various aspects of the configuration language and provides a language summary. The syntax of the language is given at the end of this appendix. A description of how to use the language can be found in the '*User Guide*'.

F.1 Introduction

The network configuration language is a special purpose language that allows linked object files to be connected to other linked object files and placed on any physical arrangement of transputers. The language has been designed to be compatible with the toolsets and allows linked object files from these toolsets to be mixed on the same network.

The main features of the language are listed below.

- The language is C-like. Declarations and expressions use C notation.
- Software and hardware networks are described using a common syntax.
- Identifiers have global scope (except replication counters).
- Arrays can be declared of any symbolic element, including processes, channels, and edges.
- Replicative and conditional statements allow easy declaration of regular networks and exceptions within them.
- New node types can be defined.
- Source files can be included.
- Comments can be inserted at any point.

A formal description of the language can be found in section F.16. The following sections describe the main features of the language and explain each of the language statements.

F.2 Statements

All statements, except the conditional statement `if` and the replication statement `rep`, must be terminated by a semicolon. Blocks of statements must be enclosed in braces '{' and '}'.

Indentation may be used to indicate structure in conditional and replicative statements, but is not required by the syntax.

F.3 Comments

Comments can appear anywhere in the configuration text and may extend over any number of lines. Comments must be preceded by the character sequence '/*' and followed by the character sequence '*/'. For example:

```
process worker; /* declare s/w process "worker" */
```

Comments may be nested.

F.4 Identifiers

Identifiers are symbolic names for configuration elements such as processors, processes, channels, edges and constants (see section F.6). Identifiers can be associated with a *type* (see below) in a typed statement. Identifiers may be followed by dimensions to create array types.

F.4.1 Character set

Identifiers can contain any letter, digit, or the underscore character; they must begin with a letter or underscore. All characters in the name are significant and letters are case-sensitive.

F.5 Types

The following base types are defined in the language:

`node` `input` `output` `edge` `connection` *numeric type*

Hardware and software networks are described as collections of interconnected **nodes**. Each node has a set of attributes defined by the node type. The node types `processor` (hardware) and `process` (software) are predefined from a defaults file that is read when the configurer is invoked on the configuration description file.

The following numeric types are available:

```
char  int  float  double
```

`char` represents the signed 8–bit integer value of a character's ASCII code. `int` is a signed 32-bit integer, `float` is a 32-bit IEEE 754 single length real, and `double` is a 64-bit IEEE 754 double length real. Types for constants are implied by the form of the numeric value.

F.6 Constants

Numeric and character constants can be defined using the `val` statement. The type of the constant will be deduced from the expression. For example:

```
val gridsize 4;   /* integer assumed */
val x_coord 2.0f; /* single length real */
val y (x*23.2e3); /* double length real expression */
```

Integers can be expressed in decimal, octal, or hexadecimal. The suffixes `K` and `M` can be used on decimal values as fixed multipliers to indicate ‘Kilo’ (2^{10}) and ‘Mega’ (2^{20}) values. Single length floating point numbers must be suffixed with the letter `F` or `f`.

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

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

Note: Any string constant that is to be passed to a C program must be explicitly terminated by the null character escape sequence `\0`. This is because the configurer does not automatically terminate strings with `\0`.

Escape sequences are defined in appendix F.16.4.

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

```
val pow2 {1, 2, 4, 8, 16, 32, 64, 128};
val powers {{1, 1, 1}, {2, 4, 8}, {3, 6, 9}};
```

F.7 Booleans

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

F.8 Expressions and arithmetic

Expressions follow the syntax used in the C programming language. Operator precedence determines the order of evaluation, and brackets can be used to override the normal ordering.

The operators which are supported are as follows:

Unary: `+ – ! ~ (`*numeric-type*`)`

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

Ternary: `?:`

All integer arithmetic is carried out to 32-bit precision. Casts can be to any of the numeric types `int`, `char`, `float` or `double`.

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

F.9 Arrays

Arrays can be declared for any base type or user-defined node type. For all array declarations except constant arrays, the dimensions are specified after the array name using the square bracket convention for subscripts. Subscripts are numbered from zero and values are stored in row order.

For constant array declarations all elements must be of the same type. In multidimensional constant arrays the dimension sizes of all the subarrays must be the same.

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

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

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

```
processor grid[4];
process slave[4];
```

F.10 Conditional statement

The `if...else` statement controls the execution of the statement that immediately follows it. The syntax of the statement is as follows:

`if` *exp statement* [`else` *statement*]

where: *exp* is any valid expression;

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

`if` can be used to conditionally place a process on a specific processor, for example, to place a process on a remote processor in a network if its memory requirements exceed a pre-determined threshold:

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

F.11 Replication

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

`rep` has two syntactic forms in which the number of iterations can be specified by a range or by an initial value followed by a count:

`rep` *index* = *exp1* `to` *exp2 statement*

`rep` *index* = *exp1* `for` *exp2 statement*

For example the following are equivalent:

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

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

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

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

The following example connects four T425 transputers in a square array and places the same process on each. The processors are connected to their neighbors via links 2 and 3; links 0 and 1 of the processor are left unconnected:

```
T425(memory=1M) grid[4];

rep i = 0 to 3
  connect grid[i].link[2] to grid[(i+1)%4].link[3];

process slave[4];

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

F.12 Built–in functions

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

F.13 Nodes

Software and hardware networks are defined using a common syntax based on the declaration of nodes and their connections. Nodes are a generic network type from which hardware and software nodes can be defined. Node types `processor` and `process` are predefined in the configurer startup file.

Nodes are associated with a number of *attributes*,the exact number and nature of which depends on the value given to the `element` attribute of a node i.e. `process` or `processor`. Nodes with element type `processor` include attributes such as `type` and `memory`, whereas nodes of type `process` have a set of runtime process attributes such as the process interface parameters, priority of execution, memory requirements, and code segment ordering.

Definitions of software nodes (processes) and hardware nodes (processors) which are the basic elements from which software and hardware networks are constructed, are read from a file of predefinitions when the configurer is invoked. The predefinitions are used as though they are defined in the language and are listed in section F.18.2.

F.13.1 Node attributes

Node attributes can be accessed in expressions using the dot convention. For example, they can be used to control the placing of processes:

```
if (master.heapsize >= 2M)
  place master on remote;
else
  place master on root;
```

When assigning values to sub-attributes open and closed brackets '`()`' are used in place of the dot. For example, the statements:

```
process(order(code=...));
process.order(code=...);
process(code=...);
```

are equivalent. (Where *process* is the name of a declared process or process type).

Sub-attributes can be specified without specifying the parent attribute. However, because some attributes have sub-attributes of the same name, ambiguity can be a problem. For example, the statements:

```
process(location(code=...));
process.location(code=...);
```

are equivalent, however, the statement:

```
process(code=...);
```

is the same as:

```
process(order(code=...));
```

The `order` and `location` attributes associated with the `process` node type have the same sub-attributes. In cases where the parent attribute (i.e. `order` or `location`) is not specified, the configurer will assume that an `order` attribute is intended.

Process and processor attributes are described in detail in the '*User Guide*', for those toolsets which support configuration.

F.13.2 Defining new node types

Refinements of existing node types can be created by using the `define` statement to specify nodes with specific attributes. As an example, consider the definition of the node types `process` and `processor` which are in the configurer startup file:

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

Once defined, node types can be used to define other node types. For example, the base type `process` can be used to define a specific process definition in the following way:

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

Or, for example, the `processor` type can be used to define a specific transputer type which can then be refined into a TRAM definition:

```
define processor(type = "T425") T425;
define T425(memory = 1M) B411;
```

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

```
workpackage slave[4];
B411 root;
T425 worker;
```

F.13.3 Connections

Nodes are connected by the `connect` statement which can be used to join software channels (unidirectional), transputer links (bi-directional), or network edges (bi-directional for hardware and uni–directional for software). The statement has two syntactic forms:

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

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

Connections can also be named for later use in the configuration, using the name of the connection defined by the optional `by` clause.

F.13.4 Prohibited connections

The following connections are disallowed and generate a configurer error:

- Inputs to inputs (except channel to edge connections)
- Outputs to outputs (except channel to edge connections)
- Processes to processors
- Network edge to network edge

F.14 Configuration language summary

F.14.1 Network data types

`node`	A point in a software or hardware network. Has the general attribute `element` (defined as '`process`' for a software network and '`processor`' for a hardware network) and other attributes defined by the value given to `element`.
`connection`	A named connection between links or channels.
`edge`	Declares a hardware network edge.
`input`	Declares a software process *input* channel or edge.
`output`	Declares a software process *output* channel or edge.

F.14.2 Numeric data types

`int`	Integer type.
`char`	Character type.
`float`	Single length floating point type (IEEE 754).
`double`	Double length floating point type (IEEE 754).

F.14.3 Language constructs

`if`	`if` *exp statement* `[else` *statement]* Simple conditional construct. *exp* can be any valid integer expression and *statement* can be a single statement or a group of statements. `else` *statement* is optional.
`rep`	`rep` *index* `=` *exp1* `to` *exp2 statement* `rep` *index* `=` *exp1* `for` *exp2 statement* Simple replication construct. Can be controlled by a range or a count.

`connect`	`connect` *item* `to` *item [*`by` *connection]*`;`
	`connect` *item*`,` *item [*`by` *connection]*`;`
	Joins channels to channels, links to links, channels to software edges, and links to hardware edges. `by` *connection* is optional.

`place`	`place` *process* `on` *processor*`;`
	`place` *channel* `on` *edge*`;`
	`place` *channel-connection* `on` *link-connection*`;`
	Assigns a software process to a processor, a channel to a link, a software edge to a hardware edge or a named channel connection to a named link connection.

`use`	`use` *filename* `for` *process*`;` Assigns a linked object file to a process.

F.14.4 Definitions and declarations

`val`	`val` *identifier exp*; Defines a numeric constant. The type is deduced from the type of the expression.
`define`	`define` *type [*`(`*attributes*`)`*] identifier*`;` Declares a node type. A list of attributes is optional.
Declare node	*type [*`(`*attributes*`)`*] name*`;` Declares a node. A list of attributes is optional.
Set attributes	*name [*`(`*attributes*`)`*]*`;` Modifies or declares attributes for a node. A list of attributes is optional.

F.14.5 Operators

Unary	`+ - ! ~ (`*numeric-type*`)`
Binary	`+ - * / % &	^ << >> && || < > <= >= == !=`
Ternary	`?:`

F.14.6 Predefinitions

Constants

Constant	Value	Use
`HIGH`, `high`	0	Indicates a high priority process.
`LOW`, `low`	1	Indicates a low priority process.
`TRUE`, `true`	1	Logical value.
`FALSE`, `false`	0	Logical value
`MIN_COST` `min_cost`	1	For use by the `routecost` processor attribute.
`MAX_COST` `max_cost`	1000000	For use by the `routecost` processor attribute.
`DEFAULT_COST` `default_cost`	1000	For use by the `routecost` processor attribute.
`INFINITE_COST` `infinite_cost`	1000001	For use by the `routecost` processor attribute.
`ZERO_TOLERANCE` `zero_tolerance`	0	For use by the `tolerance` processor attribute.
`DEFAULT_TOLERANCE` `default_tolerance`	1	For use by the `tolerance` processor attribute.
`MAX_TOLERANCE` `max_tolerance`	1000000	For use by the `tolerance` processor attribute.
`ROUTER_ORDER` `router_order`	–20000	Weights the relative position of software virtual routing system processes in memory.
`MUXER_ORDER` `muxer_order`	–10000	Weights the relative position of software multiplexing system processes in memory.

Types

process PROCESS	Software process node type.
lowprocess LOWPROCESS	Software low priority process node type.
highprocess HIGHPROCESS	Software high priority process node type.
processor PROCESSOR	Hardware processor node type.
T212 t212 T222 t222 T225 t225 M212 m212	IMS T2 processor series.
T400 t400 T414 t414 T425 t425 T450 t450	IMS T4 processor series.
T800 t800 T801 t801 T805 t805	IMS T8 processor series.
ST20 st20 ST20_1 ST20_2 ST20_3 ST20_4 st20_1 st20_2 st20_3 st20_4	The ST20 processor series.

F.14.7 Built–in functions

size	Returns the size of an array.

F.15 Configurer directives

F.15.1 #include

This directive allows the configuration source to be split between several files.

```
#include "filename"
```

The configuration source file *filename* will be included at the line where this directive is inserted.

F.15.2 Configurer directives summary

#include	#include *"filename"* Includes another source file.

F.16 Configuration language syntax

The section gives details of the configuration language syntax.

F.16.1 Notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly, the form is as follows:

- Terminal strings of the language — those not built up by rules of the language — are printed in teletype font e.g. `node`.
- Each phrase definition consists of an equality expression built up using a double colon and an equals sign to separate the two sides e.g. '*::=*'.
- Alternatives are separated by vertical bars ('|').
- Optional sequences are enclosed in square brackets ('*[*' and '*]*').
- Items which may be repeated zero or more times appear in braces ('*{*' and '*}*').
- $\{_0, x\}$ represents a list of zero or more items of type 'x' separated by commas.
- $\{_1, x\}$ represents a list of one or more items of type 'x' separated by commas.

F.16.2 Configuration

```
configuration      ::=    config-item {config-item}

config-item        ::=    declaration
                       |  replicator
                       |  conditional
                       |  directive

declaration        ::=    node-decl
                       |  node-attr-decl
                       |  nodedef-decl
                       |  connect-decl
                       |  edge-decl
                       |  connector-decl
                       |  mapping-decl
                       |  numeric-value-decl
                       |  compound-decl
                       |  use-decl

compound-decl      ::=    { config-item {config-item} }
```

F.16.3 Language features

```
letter              ::=   A | B | ... | Z | a | b | ... | z

digit               ::=   0 | 1 | 2 | ... | 9

id-char             ::=   letter | digit | _

identifier          ::=   letter {id-char}
                        | _ {id-char}

comment             ::=   /* any characters except */ sequence */

directive           ::=   file-include

file-include        ::=   #include string
```

F.16.4 Expressions

```
octal-digit         ::=   0 | 1 | 2 | ... | 7

hex-digit           ::=   digit | A | B | ... | F | a | b | ... | f

octal               ::=   0 octal-digit {octal-digit}

decimal             ::=   digit {digit}

hex                 ::=   0x hex-digit {hex-digit}
                        | 0X hex-digit {hex-digit}

character-const     ::=   ' char '

char                ::=   any character except end of line and '
                        | escape-sequence

escape-sequence     ::=   \'  | \"  | \\  | \?
                        | \a | \b | \f | \n | \r | \t | \v
                        | \ octal-digit [octal-digit] [octal-digit]
                        | \x {hex-digit}

string              ::=   " {string-char} "

string-char         ::=   any character except end of line and "
                        | escape-sequence

scale-size          ::=   k | K | l | L

int-const           ::=   decimal[scale-size]
                        | octal
                        | hex
```

```
sign              ::=    + | -

exponent          ::=    E [sign] decimal
                     |   e [sign] decimal

real-size         ::=    f | F | l | L

real-const        ::=    decimal . [decimal] [exponent] [real-size]
                     |   decimal exponent [real-size]
                     |   . decimal [exponent] [real-size]

array-const       ::=    [ {1, exp } ]
                     |   string

subscript         ::=    [ exp ]  { [ exp ] }

const             ::=    int-const
                     |   real-const
                     |   character-const
                     |   array-const [subscript]

numeric-type      ::=    int | float | double | char

monadic-op        ::=    + | - | ! | ~
                     |   ( numeric-type )

dyadic-op         ::=    + | - | * | / | %
                     |   & | | | ^ | << | >>
                     |   && | ||
                     |   < | > | <= | >= | == | !=

element           ::=    identifier { [subscript] . identifier } [subscript]

function-call     ::=    size ( element )

exp               ::=    const
                     |   element
                     |   monadic-op exp
                     |   exp dyadic-op exp
                     |   exp ? exp : exp
                     |   ( exp )
                     |   function-call
```

F.16.5 Replication and conditionals

```
replicator        ::=    rep identifier = exp to exp  config–item
                     |   rep identifier = exp for exp  config–item

conditional       ::=    if exp config–item [else config–item]
```

F.16.6 Numeric value declarations

```
numeric-value-decl  ::=     val identifier exp ;
```

F.16.7 Network declarations

```
node-decl        ::=  node-type [( {1, attributes} )] {1, identifier [subscript]} ;

node-type        ::=     node
                       | identifier

attributes       ::=     general-attr
                       | node-attr
                       | process-attr
                       | processor-attr

general-attr     ::=     identifier = exp
                       | identifier ( {1, general-attr} )
                       | identifier ( {1, formal-attr} )

node-attr        ::=     element = element-type

element-type     ::=  "processor"
                 |    "process"

processor-attr   ::=  type = processor-type
                 |    memory = exp
                 |    memstart = exp
                 |    numlinks = exp
                 |    reserved = exp
                 |    router ( {1, router-attr} )

router-attr      ::=  linkquota = exp
                 |    routecost = exp
                 |    tolerance = exp

processor-type   ::=  "ST20"
                 |    "M212"
                 |    "T212"
                 |    "T222"
                 |    "T225"
                 |    "T400"
                 |    "T414"
                 |    "T425"
                 |    "T450"
                 |    "T800"
                 |    "T801"
                 |    "T805"
```

```
process-attr      ::=     stacksize = exp
                        | heapsize = exp
                        | priority = exp
                        | nodebug = exp
                        | noprofile = exp
                        | interface ( {1, formal-att} )
                        | order ( {1, segment-att} )
                        | location ( {1, segment-att} )

segment-attr      ::=     code = exp
                        | heap = exp
                        | stack = exp
                        | static = exp
                        | vector = exp

formal-attr       ::=     formal-type {1, identifier [subscript]  [= exp] }

formal-type       ::=     numeric-type
                        | input
                        | output

node-attr-decl    ::=     element  [( {1, attributes } )] ;

nodedef-decl      ::=     define node-type  [( {1, attributes } )]  identifier ;

connector-decl    ::=     connection   {1, identifier [subscript] } ;

connect-decl    ::=  connect element , element  [by identifier [subscript]] ;
                   | connect element to element  [by identifier [subscript]] ;

edge-decl         ::=  edge {1, identifier [subscript] } ;
                   |   input {1, identifier [subscript] }
                   |   output {1, identifier [subscript] } ;
```

F.16.8 Mapping declarations

```
mapping-decl      ::=  place element on element ;

use-decl          ::=  use string for element ;
```

F.17 Implementation details

- Subscript ranges for arrays are dependent on the word length of the machine running the configurer. For 16-bit machines the range is 0 to 2^{15}–1, for 32-bit machines the range is 0 to 2^{31}–1.
- Each line in the source configuration file should not exceed 1024 characters, not including leading and following white space.
- The maximum number of dimensions for an identifier or array constant is 64. If this limit is exceeded then an error message will be generated. (This also means that any occam process referenced in the configuration description cannot have a formal parameter that has more than 64 dimensions).

F.18 Reserved words

This section defines the set of reserved words, and predefined types, attributes and constants, that are defined in the configuration language.

F.18.1 Keywords

Reserved words cannot be used by the programer as identifiers in the configuration description.

The reserved words are as follows:

```
by          char        connect     connection
define      double      edge        else
float       for         if          input
int         node        on          output
place       rep         size        to
use         val
```

F.18.2 Pre–defined attributes

Node attributes

The `element` attribute used for defining the type of a `node` can take the following values:

- `process` – the node is a *process* in a *software* network.
- `processor` - the node is a *processor* in a *hardware* network.

Note: The names of node attributes are not reserved words and can be freely used as general purpose identifiers by the programmer.

Processor attributes

The attributes defined for nodes of type `processor` are as follows:

- `link` - used by processor nodes to define interconnection. Only defined if the `type` attribute has already been defined. In addition if `type` takes a value of `ST20` then `link` will only become defined when the `numlinks` attribute is defined.

- `memory` - used by processor nodes to define memory size.

- `memstart` - specifies the location of **MemStart** as an absolute address which must be word aligned. Only supported for processors whose `type` attribute is set to `ST20`.

- `numlinks` - defines the number of links available on a processor and can take the values `1` to `4` inclusive. This attribute only applies when `type` is set to `ST20`. `numlinks` must be defined, in order for a value for `link` to be determined.

- `reserved` - used by processor nodes to reserve memory.

- `router` - used by processor nodes to control the virtual routing decisions of the configurer. The `router` attribute can take the following sub–attributes: `linkquota`, `routecost` and `tolerance`.

    - `linkquota` - suggests the maximum number of links on the associated processor that should be used by the virtual channel routing system.

    - `routecost` - defines within the range `MIN_COST` to `MAX_COST` inclusive the associated cost of through-routing data through this processor for other processor's virtual channel traffic.

    - `tolerance` - controls with any value between `ZERO_TOLERANCE` and `MAX_TOLERANCE` inclusive how much the particular processor concerned can be used for the provision of load-sharing through-routing paths for other processors.

- `type` - used by processor nodes to define processor type. Processor types predefined in standard include files are as follows:

```
ST20
T400    T414    T425    T450
T800    T801    T805
T212    T222    T225
M212
```

Process attributes

The attribute names currently defined for nodes of type process are:

- heapsize – used by the process nodes to specify the size of the heap data segment used by the process.
- interface – used by process nodes to define the type and the default values of parameters to be passed into the process when the process starts executing.
- location – used by process nodes to specify the absolute locations of their code and data segments. The location attribute can take the following sub-attributes:

  ```
  code   stack   static   heap   vector
  ```

- nodebug – used by process nodes to notify the INQUEST debugger whether the process is to be debugged or not.
- noprofile – used by process nodes to notify the INQUEST profiler whether the process is to be profiled or not.
- order – used by process nodes to specify the ordering of their code and data segments. The order attribute can take the following sub-attributes:

  ```
  code   stack   static   heap   vector
  ```

- priority – used by process nodes to specify the priority of the process.
- stacksize – used by the process nodes to specify the size of the stack data segment used by the process.

F.19 Predefinitions

The following definitions are read by the configurer from the standard include file setconf.inc at invocation.

F.19.1 Constants

```
val FALSE 0;
val TRUE 1;

val false 0;
val true 1;

val HIGH 0;
val LOW 1;

val high 0;
val low 1;
```

```
val MIN_COST                      1;
val DEFAULT_COST                  1000;
val MAX_COST                      1000000;
val INFINITE_COST      1000001;

val ZERO_TOLERANCE     0;
val DEFAULT_TOLERANCE 1;
val MAX_TOLERANCE      1000000;

val ROUTER_ORDER                  -20000;
val MUXER_ORDER                   -10000;

val min_cost                      1;
val default_cost                  1000;
val max_cost                      1000000;
val infinite_cost      1000001;

val zero_tolerance     0;
val default_tolerance 1;
val max_tolerance      1000000;

val router_order                  -20000;
val muxer_order                   -10000;
```

`TRUE`, `true`, `FALSE`, and `false` are used in expressions where a boolean value is needed.

`HIGH`, `high`, `LOW`, and `low` can be used to define the execution priority for a process.

The remaining constants are used to influence the performance of the software virtual routing network.

F.20 Types

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

define node (element = "processor", type = "ST20") st20;
define node (element = "processor", type = "ST20", numlinks = 1) st20_1;
define node (element = "processor", type = "ST20", numlinks = 2) st20_2;
define node (element = "processor", type = "ST20", numlinks = 3) st20_3;
define node (element = "processor", type = "ST20", numlinks = 4) st20_4;
define node (element = "processor", type = "T805") t805;
define node (element = "processor", type = "T801") t801;
define node (element = "processor", type = "T800") t800;
define node (element = "processor", type = "T450") t450;
define node (element = "processor", type = "T425") t425;
define node (element = "processor", type = "T414") t414;
define node (element = "processor", type = "T400") t400;
define node (element = "processor", type = "T225") t225;
define node (element = "processor", type = "T222") t222;
define node (element = "processor", type = "T212") t212;
define node (element = "processor", type = "M212") m212;
define node (element = "processor") PROCESSOR;

define node (element = "processor", type = "ST20") ST20;
define node (element = "processor", type = "ST20", numlinks = 1) ST20_1;
define node (element = "processor", type = "ST20", numlinks = 2) ST20_2;
define node (element = "processor", type = "ST20", numlinks = 3) ST20_3;
define node (element = "processor", type = "ST20", numlinks = 4) ST20_4;
define node (element = "processor", type = "T805") T805;
define node (element = "processor", type = "T801") T801;
define node (element = "processor", type = "T800") T800;
define node (element = "processor", type = "T450") T450;
define node (element = "processor", type = "T425") T425;
define node (element = "processor", type = "T414") T414;
define node (element = "processor", type = "T400") T400;
define node (element = "processor", type = "T225") T225;
define node (element = "processor", type = "T222") T222;
define node (element = "processor", type = "T212") T212;
define node (element = "processor", type = "M212") M212;

define node (element = "process") process;
define node (element = "process", priority = high) highprocess;
define node (element = "process", priority = low) lowprocess;

define node (element = "process") PROCESS;
define node (element = "process", priority = HIGH) HIGHPROCESS;
define node (element = "process", priority = LOW) LOWPROCESS;
```

F.21 Declarations

One declaration is defined in `setconf.inc`:

```
edge host;
```

G occam configuration language

This appendix describes various aspects of the occam configuration language and defines the syntax of the occam configuration language. This should be considered as extending the syntax of occam. A full description of how to use the language can be found in the 'Configuration' chapter in the '*User Guide*'.

G.1 Notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly, the form is as follows:

- Each phrase definition consists of an equality expression built up using an equals sign to separate the two sides.
- Terminal strings of the language — those not built up by rules of the language — are printed in teletype font e.g. `NODE`.
- Alternatives are separated by vertical bars ('|').
- Optional sequences are enclosed in italic square brackets ('*[*' and '*]*').
- Items which may be repeated zero or more times appear in braces ('{' and '}').
- $\{_0, x\}$ represents a list of zero or more items of type 'x' separated by commas.
- $\{_1, x\}$ represents a list of one or more items of type 'x' separated by commas.

G.2 Introduction

The occam 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. The configuration language consists of the following:

- a hardware description
- a software description
- an optional mapping description

The mapping may appear either before of after the software configuration, but after the declaration of any nodes and after the hardware description. Normal occam scope rules apply.

The `#INCLUDE` mechanism may be used to incorporate hardware descriptions, software descriptions, or any source text from other files.

The #USE statement may be used to reference pre-compiled code, either at the outer level, or within the software description.

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 G.1.

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

Table G.1 Configuration description declarations

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

Configuration declarations are usually followed by statements which perform various actions relating to the declaration. Actions are defined by SET, CONNECT and MAP statements. The DO construct enables these statements to be grouped or replicated. PROCESSOR statements introduce processes which may be mapped onto named processors. IF may be used as in occam. Configuration language statements are listed in Table G.2.

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

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

Table G.2 Configuration description statements

Importing code and source files

Compiled code from external 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. For example, the software description and the mapping description may be kept in different files, accessed by #INCLUDE directives from a 'master' configuration description file.

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

G.3 New types and specifications

This section defines the new occam types introduced by the configuration language.

The syntax adds the new primitive types NODE, EDGE and ARC, and structures CONFIG, NETWORK and MAPPING to the occam language.

NODE declarations introduce processors (*nodes* of a graph). These processors are *physical* if their type and memory size attributes are defined as part of the hardware description, and *logical* otherwise.

EDGE declarations introduce external connections of the hardware description.

ARC declarations introduce named connections (*arcs* of a graph). Each arc connects two edges, which may be attributes of nodes, or declared edges. Connections need only be named if it is required to force a particular mapping of channels, or if names are required to aid debugging.

NETWORK declarations introduce the hardware description.

CONFIG declarations introduce the software description.

MAPPING declarations introduce the mapping description.

G.3.1 Syntax of configuration description

```
configuration          =   hardware.description
                           software.description
                           [ mapping ]
                       |   specification
                           configuration

specification              VAL...  (see the 'occam 2 Reference Manual')

primitive type         =   NODE
                       =   EDGE
                       =   ARC
```

G.4 Hardware description

The NETWORK keyword introduces a hardware description, an optionally named structure which describes the types, connectivity and attributes of previously declared processor nodes. Connections are defined in CONNECT statements. Attributes are given values in SET statements. The attributes of a processor node include an array of edges which are its links, a string which defines its processor type, and an integer value which is the memory size in bytes.

Connections and attribute settings may be combined in any order using the DO constructor, including replication and conditionals. For each processor node the attributes with predefined names type and memsize must be set once (and only once). The connections connect declared edges and edges of nodes, which have the predefined attribute name link. The predefined boolean attribute root may be set to TRUE for only one node in a network without a connection to the predefined edge HOST. The predefined attribute romsize defines the size in bytes of read only memory on a node. Attributes are referenced by subscripting node names with attribute names in square brackets ([]).

G.4.1 Processor attributes

This section describes processor attributes defined in the occam configuration language.

The following processor attributes can be defined in the NETWORK description:

- link – used by processor nodes to define interconnection. Only defined if the type attribute has already been set.
- memsize – used by processor nodes to define memory size.
- root – defines the root processor if there is no host connection.
- romsize – specifies the size of ROM attached to the root processor, expressed as an integer value.
- type – used by processor nodes to define their processor type.

The following optional processor attributes can be defined in the MAPPING description:

- linkquota – suggests the maximum number of links on the associated processor that should be used by the software virtual channel routing system.
- location.code, location.ws, location.vs, – used by processor nodes to specify the absolute locations of their code, workspace, and vectorspace segments.
- order.code, order.ws, order.vs – used to specify the ordering of code and data segments on a processor node.
- nodebug – disables debugging by the *Inquest Toolset.* Takes the value TRUE or FALSE. The default is FALSE.
- noprofile – disables profiling by the *Inquest Toolset.* Takes the value TRUE or FALSE. The default is FALSE.
- reserved – used by processor nodes to reserve memory from MOSTNEG INT.
- routecost – defines within the range 1 to 1000000 the associated cost of software through-routing data through this processor for other processor's virtual channel traffic.
- tolerance – controls with any value between 0 and 1000000 how much the particular processor concerned can be used by the software virtual channel routing system for the provision of load-sharing through-routing paths for other processors.

G.4.2 Syntax definition

```
hardware.description     =   NETWORK [ network.name ]
                                network.item
                             :
                         |   specification
                             hardware.description

specification            =   node.declaration
                         |   edge.declaration
                         |   arc.declaration
                         |   VAL...  (see the 'occam 2 Reference Manual')

node.declaration         =   { [ expression ] }NODE node.name :
edge.declaration         =   { [ expression ] }EDGE edge.name :
arc .declaration         =   { [ expression ] }ARC arc.name :

network.item             =   DO [ replicator ]
                                network.item
                         |   connection.item
                         |   attribute.setting
                         |   conditional.network.item
                         |   SKIP
                         |   STOP
                         |   abbreviation
                             network.item

connection.item          =   CONNECT edge TO edge [WITH arcname ]

edge                     =   edge.name { [subscript] }
                         |   node [link]  [expression]

attribute.setting        =   SET node ( attribute.assignment )
node                     =   node.name { [ subscript ] }
attribute.assignment     =   {1 , attribute } :=  {1 , attribute.value }
attribute                =   attribute.name { [ subscript ] }
attribute.value          =   expression

conditional.network.item =   IF
                               {network.choice}

network.choice           =   guarded.network.choice
                         |   conditional.network.item

guarded.network.choice   =   boolean
                               network.item
```

G.5 Software description

A CONFIG declaration introduces the software description as an occam process. Additional specifications and processes are added to occam: the processor name in a PROCESSOR statement which may be a physical processor name or the name of a logical processor which is mapped onto a physical processor. A channel allocation may allocate channels onto a named arc of the network. Arcs and physical processors are defined in the hardware description.

G.5.1 Syntax definition

```
software.description      =   CONFIG [config.name]
                                 placedpar
                              :
                          |   specification
                              software.description

placedpar                 =   [PLACED] PAR [ replicator]
                                 placedpar
                          |   PROCESSOR node
                                process (see the 'occam 2 Reference Manual')
                          |   specification
                              placedpar
                          |   conditional.network.item
                          |   SKIP
                          |   STOP

conditional.network.item  =   IF
                                 {network.choice}

network.choice            =   guarded.network.choice
                          |   conditional.network.item

guarded.network.choice    =   boolean
                                 placedpar

channel.allocation        =   PLACE {channel.list} ON arc:

node                      =   node.name { [ subscript ] }

arc                       =   arc.name { [ subscript ] }
```

G.6 Mapping structure

The MAPPING keyword introduces an (optionally named) mapping structure which may be either before or after the software description. The IF construct may be used as in occam. SKIP and STOP are also allowed. A DO construct facilitates replication.

Mappings are introduced by the MAP keyword. A mapping may be used to associate logical processors with physical processors (code mapping), and channels with arcs (channel mappings).

In code mappings a logical processor may appear on the left hand side of only one mapping item whereas a physical processor may appear on the right hand side of more than one mapping item. A code mapping may include a priority clause, introduced by PRI, which will determine the priority at which the process will run. SET may be used to define processor attributes, see section G.4.1.

Channel mappings are optional except in the case where one end of the *arc* is an external edge. In channel mappings the arc must connect the nodes onto which the processes using the channels are mapped. The effect of channel mappings is identical to the corresponding channel allocations which may appear in the software description.

G.6.1 Syntax definition

```
mapping                  =   MAPPING [ mapping.name ]
                                mapping.item
                             :
                         |   specification
                             mapping

mapping.item             =   code.mapping
                         |   channel.mapping
                         |   channel.allocation
                         |   DO  [ replicator ]
                                mapping.item
                         |   attribute.setting
                         |   conditional.map.item
                         |   SKIP
                         |   STOP
                         |   abbreviation
                             mapping.item


code.mapping             =   MAP node.list ONTO node [priority.clause]
priority.clause          =   PRI expression
node.list                =   {1 , node }
node                     =   node.name { [ subscript ] }

channel.mapping          =   MAP channel.list ONTO arc
channel.list             =   {1 , channel }
channel                  =   channel.name { [ subscript ] }
arc                      =   arc.name { [ subscript ] }

channel.allocation       =   PLACE { channel.list } ON arc:

attribute.setting        =   SET node ( attribute.assignment )
attribute.assignment     =   {1 , attribute } :=  {1 , attribute.value }
attribute                =   attribute.name { [ subscript ] }
attribute.value          =   expression

conditional.map.item     =   IF
                               {mapping.choice}

mapping.choice           =   guarded.mapping.choice
                         |   conditional.map.item

guarded.mapping.choice   =   boolean
                              mapping.item
```

G.7 Constraints

The following constraints apply to all configurations:

- All physical processors whose `types` are set must be connected to each other.
- Any physical processor whose `type` is set must have its `memsize` set.
- Logical processors may only be mapped onto physical processors whose `type` has been set.
- Channels connecting processors of different word size must not use protocols based on the type `INT`.
- A priority expression must evaluate to 0 (high) or 1 (low).
- The configuration tool will reject the mapping at high priority of a process which itself includes a `PRI PAR`.
- Any channel which is explicitly placed onto an arc can only be used by processors which directly connect to that arc.
- The configurer is applied to a program *after* it has been linked; this means that files referenced via the `#USE` statement must have already been linked. It also means that you may not access libraries from the software description, except for libraries of linked units.

  A side effect of this is that some arithmetic which would normally be performed via calls to compiler libraries will be disallowed within the software description.
- Channel abbreviations referencing replicator variables are not permitted outside a `PROCESSOR` construct. For example, the first code extract below must be changed to the second:

```
CONFIG
  PAR i = 0 FOR n
    inchan IS chan[i] :
    outchan IS chan[i+1] :
    PROCESSOR p[i]
      process(inchan, outchan)
:
```

```
CONFIG
  PAR i = 0 FOR n
    PROCESSOR p[i]
      inchan IS chan[i] :
      outchan IS chan[i+1] :
      process(inchan, outchan)
:
```

- `CHAN` declarations are not permitted inside replicators but outside `PROCESSOR` constructs.
- The configurer can compile a maximum of 40K of ‘object’ code for any single processor. This is all the code within the `PROCESSOR` statements for each processor. (This limit can be changed by using a command line switch).
- `INT` array constructors such as `[1,2,3]` are not accepted for 16-bit processors. They should be converted into `INT16` arrays.
- It is not permitted to `RETYPE INT` constants into other types for 16-bit processors. `INT16` constants should be used instead.
- `INT` expressions are treated as `INT32`s. Thus `MOSPOS INT` evaluates to the same value as `MOSTPOS INT32`. Where this is a problem, i.e. it causes a 16-bit integer overflow, the configurer will generate an error.

G.8 Checking IF statements

The configurer checks subscripts in all expressions during configuration, including those inside code which will never be executed. Hence the following code is illegal:

```
VAL n IS 1 :
[n]CHAN OF INT c :
CONFIG
  IF
    n > 1
      PROCESSOR proc
        p(c[1])        -- illegal because n = 1
    TRUE
      PROCESSOR proc
        p(c[0])
:
```

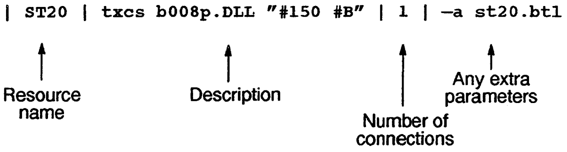
H AServer database

This appendix describes the AServer database, its purpose, use and format. The AServer database replaces the connection database used with earlier Toolsets. The environment variable or Windows environment file variable `ASERVDB` should hold the full path for the AServer database file if it is not on the `ISEARCH` path. An example AServer database is supplied with the interface software.

An AServer database 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.*

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 H.1. The entries are case sensitive, so, for example, `ST20` is not the same as `st20`.



```
| ST20 | txcs b008p.DLL "#150 #B" | 1 | -a st20.btl
```

Figure H.1 AServer database line

H.1 Target hardware connection fields

The four fields in a line defining a target hardware connection are:

1. The name of the resource, such as `ST20`. If the resource is a connection to target hardware then the target may be used by:

   ○ setting the environment variable or Windows environment file variable `TRANSPUTER` to this name or

   ○ using the name with the `irun` option `SL`, such as:

   ```
   irun -sl ST20 app.btl
   ```

   If the resource is an AServer service then the name may be used by the client to request the service.

2. A description of the resource, such as `txcs b008p.DLL "#150 #B"`. This description tells the interface software how to communicate with the target. The format of this description depends on the interface software being used. The description consists of:

   - an AServer control service (e.g. `txcs`);
   - the driver for this type of interface (e.g. `b008p.DLL`);
   - the parameter string for the interface driver.

   For further details see the '*ST20 Toolset Delivery Manual*'.

3. The number of connections that may be made to the resource or to a single instance of the service. For target hardware connections this is always 1.

4. Any extra parameters that may be needed. For AServer services, this may be any extra parameters needed in the service command line. For ST20 hardware booting from link, this should include `–a` followed by the memory interface file name.

I ITERM files

This appendix describes the format of ITERM files; it is included for people who need to write their own ITERM because they are using terminals that are not supported by the standard ITERM file supplied with the toolset.

Standard ITERM files for this release are provided in the `iterms` directory, which is a subdirectory of the main toolset installation directory. These files may be used as templates and tailored to suit your own needs. It is recommended that the installation files are not changed in any way, and that modifications are only made to copies of the files.

I.1 Introduction

ITERMs are ASCII text files that describe the control sequences required to drive terminals. Screen oriented applications that use ITERM files are terminal independent.

ITERM files are similar in function to the UNIX *termcap* database and describe input from, as well as output to, the terminal. They allow applications that use function keys to be terminal independent and configurable.

Within the toolset, the ITERM file is only used by the memory interface tool `imem`.

A default ITERM file may be defined in the `ITERM` environment variable. For details see section I.8 and the Delivery Manual for the release.

I.2 The structure of an ITERM file

An ITERM file consists of three sections. These are the *host*, *screen* and *keyboard* sections. Sections are introduced by a line beginning with the section letters '`H`', '`S`' or '`K`'. Case is unimportant and the rest of the line is ignored. Sections consist of a number of lines beginning with a digit. A section is terminated by a line beginning with the letter '`E`'. The *host* section must appear first; other sections may appear in any order in the file. Sections must be separated by at least one blank line.

The syntax of the lines that make up the body of a section is best described in an example:

```
3:34,56,23,7.    comments
```

Each line starts with the index number followed by a colon and a list of numbers separated by commas. Each line is terminated by a full stop ('`.`') and anything following it is treated as a comment. Spaces are not allowed in the data string and an entry cannot be split across more than one line.

Comment lines, beginning with the character '#', may be placed anywhere in an ITERM file. Extra blank lines in the file are ignored.

The index numbers in each section correspond to an agreed meaning for the data. In the following sections the meaning of the data in each of the three sections is described in detail.

I.3 The host definitions

I.3.1 ITERM version

This item identifies an ITERM file by version. It provides some protection against incompatible future upgrades.

e.g. `1:2.`

I.3.2 Screen size

This item allows applications to find out the size of the terminal at startup time. The data items are the number of columns and rows, in that order, available on the current terminal.

e.g. `2:80,25.`

Screen locations should be numbered from 0, 0 by the application. Terminals which use addressing from 1, 1 can be compensated for in the definition of goto X, Y.

I.4 The screen definitions

The lists of values in the screen section represent control codes that perform certain operations; the data values are ASCII codes to send to the display device.

ITERM version 2 defines the indices given in table I.1. These definitions are used in the example ITERM file; for a complete listing of the file see section I.8.

Index	Screen operation
1	cursor up
2	cursor down
3	cursor left
4	cursor right
5	goto x y
6	insert character
7	delete character at cursor
8	clear to end of line
9	clear to end of screen
10	insert line
11	delete line
12	ring bell
13	home and clear screen
20	enhance on (not used)
21	enhance off (not used)

Table I.1 ITERM screen operations

For example, an entry like: '`8:27,91,75.`' indicates that an application should output the ASCII sequence '`ESC [ K`' to the terminal output stream to clear to end of line.

I.4.1 Goto X Y processing

The entry for 5, 'goto X Y', requires further interpretation by the application. A typical entry for 'goto X Y' might be:

```
5:27,-11,32,-21,32
```

The negative numbers relate to the arguments required for X and Y.

..., -ab, nn, ...

where: *a* is the argument number (i.e. 1 for X, 2 for Y).

b controls the data output format.

If *b* =1 output is an ASCII byte (e.g. 33 is output as `!`).

If *b* =2 output is an ASCII number (e.g. 33 is output as `33`).

nn is added to the argument before output.

As a complete example, consider the following ITERM entry in the screen section:

```
5:27,91,-22,1,59,-12,1,72. ansi cursor control
```

This would instruct an application wishing to move the terminal cursor to X=14, Y=8 (relative to 0,0) to output the following bytes to the screen:

```
Bytes in decimal: 27   91  57  59  49  53  72
Bytes in ASCII:   ESC  [   9   ;   1   5   H
```

I.5 The keyboard definitions

Each index represents a single keyboard operation. The data specified after each index defines the keystroke associated with that operation. Multiple entries for the same index indicate alternative keystrokes for the operation.

ITERM version 2 defines the indices given in table I.2. These definitions are used in the example ITERM file; for a complete listing of the file see section I.8.

Index	Function
2	delete character
6	cursor up
7	cursor down
8	cursor left
9	cursor right
12	delete line
14	start of line
15	end of line
18	line up
19	line down
20	page up
21	page down
26	enter file
27	exit file
28	refresh
29	change file
31	finish
34	help
36	get address

Index	Function
39	goto line
40	backtrace
41	inspect
42	channel
43	top
44	retrace
45	relocate
46	info
47	modify
48	resume
49	monitor
50	word left
51	word right
55	top of file
56	end of file
62	toggle hex
65	continue from
66	toggle breakpoint
67	search

Table I.2 ITERM key operations

I.6 Setting up the ITERM environment variable

To use an ITERM the application has to find and read the file. An environment variable called `ITERM` should be set up with the pathname of the file as its value.

For more details about setting environment variables see the Delivery Manual that accompanies the release.

I.7 Iterms supplied with a toolset

The following ITERM files are supplied with the toolset:

File	Description
`ansi.itm`	Generic ANSI iterm
`ncd.itm`	NCD X terminal iterm
`necansi.itm`	NEC PC iterm
`pcansi.itm`	PC iterm (requires ANSI.SYS)
`sun.itm`	SunView iterm
`vt100.itm`	vt100 iterm

Table I.3 ITERM files supplied

`ansi.itm` is likely to be the most portable in that it will work unchanged with most hosts. However, because of this it may only use the normal (alpha–numeric keys) of a keyboard. This means that some keys (when used in conjunction with the **CNTL** or **SHIFT** key) are associated with more than one operation. Specific host iterms make use of known function keys etc. which leads to less overloading of keys.

Each iterm file may be treated as an example; you may create and use your own iterm file if you wish.

I.8 An example ITERM

This is the generic toolset ITERM file for an ANSI terminal.

```
#  --------------------------------------------------
#
#   ANSI ITERM for any ANSI terminal
#   Support for imem
#
#     V1.0 16 November 1990          (RD)  Created
#     V1.1 11  January 1991          (NH)  Modified
#
#  --------------------------------------------------

host section
1:2.                          version
2:80,24.                      screen size
end of host section

#  screen control characters

screen section
#                             DEBUGGER       SIMULATOR
1:27,91,65.                                  cursor up
2:27,91,66.                                  cursor down
3:27,91,68.                   cursor left    cursor left
4:27,91,67.                                  cursor right
5:27,91,-22,1,59,-12,1,72.    goto x y       goto x y
#6.                           insert char    insert char
#7.                           delete char    delete char
8:27,91,75.                   clear to eol   clear to eol
9:27,91,74.                   clear to eos   clear to eos
#10        ansi terminals do  insert line    insert line
#11        not have these     delete line    delete line
12:7.                         bell           bell
13:27,91,50,74.               clear screen   clear screen
end of screen section


keyboard section
#                        KEY                DEBUGGER       SIMULATOR
#
2:127.                   # DELETE           del char
2:8.                     # BACKSPACE        del char
6:27,91,65.              # UP               cursor up      cursor up
7:27,91,66.              # DOWN             cursor down    cursor down
8:27,91,68.              # LEFT             cursor left    cursor left
9:27,91,67.              # RIGHT            cursor right   cursor right
```

```
12:21.                  # Crtl-U        delete line
14:1.                   # CTRL-A        start of line  start of line
15:5.                   # CTRL-E        end of line    end of line
18:27,85.               # ESC U         line up
18:27,117.              # ESC u         line up
19:27,68.               # ESC D         line down
19:27,100.              # ESC d         line down
20:27,86.               # ESC V         page up        page up
20:27,118.              # ESC v         page up        page up
21:27,87.               # ESC W         page down      page down
21:27,119.              # ESC W         page down      page down
26:14.                  # CTRL-N        enter file
27:24.                  # CTRL-X        exit file
28:12.                  # CTRL-L        refresh        refresh
28:23.                  # CTRL-W        refresh        refresh
29:27,70.               # ESC F         change file
29:27,102.              # ESC f         change file
31:27,88.               # ESC X         finish
34:27,72.               # ESC H         help           help
34:27,104.              # ESC h         help           help
36:27,65.               # ESC A         get address
36:27,97.               # ESC a         get address
39:7.                   # CTRL-G        goto line
40:27,48.               # ESC 0         backtrace
41:27,49.               # ESC 1         inspect
41:9.                   # CTRL-I        inspect
42:27,50.               # ESC 2         channel
43:27,51.               # ESC 3         top
44:27,52.               # ESC 4         retrace
45:27,53.               # ESC 5         relocate
46:27,54.               # ESC 6         info
47:27,55.               # ESC 7         modify
48:27,56.               # ESC 8         resume
49:27,57.               # ESC 9         monitor
50:11.                  # CTRL-K        word left
51:16.                  # CTRL-P        word right
55:27,60.               # ESC <         top of file
56:27,62.               # ESC >         end of file
62:27,116.              # ESC t         toggle hex
62:27,84.               # ESC T         toggle hex
65:27,67.               # ESC C         continue from
65:27,99.               # ESC c         continue from
66:2.                   # CTRL-B        toggle break
67:6.                   # CTRL-F        search (Find)
end of keyboard section


#   THAT'S ALL FOLKS
```

Index