Parallel Fortran User Guide

3L Ltd

This edition November 30, 1990 describes version 2.1.3 of the software.

20 19 18 17 16 15 14 13 12 11
10 9 8 7 6 5 4 3 2 1

3L Ltd
Peel House
Ladywell
Livingston EH54 6AG
Scotland
Tel. (0506) 41 59 59
Fax. (0506) 41 59 44

Not For Sale

Contents

Introduction

Intended Audience

This *User Guide* accompanies 3L's Parallel Fortran product, and is intended for anyone who wants to use Parallel Fortran to program a transputer system, whether writing a conventional sequential program or using the full support for concurrency which the transputer processor has to offer.

Hardware Assumptions

Parallel Fortran can be used with a large variety of target transputer systems. This manual makes the simplifying assumption that the target hardware will be an Inmos IMS B004 transputer evaluation board, or a transputer system which is largely compatible with a B004. This board is a single plug-in card for the standard IBM PC bus, with one transputer and either 1MB or 2MB of RAM.

Similarly, the assumption is made here that the host computer for the B004 will be an IBM PC with a hard disk drive, or one of the many personal computers compatible with the original IBM machines.

Document Structure

There are five main divisions within this document, as follows:

- *Part I: Getting Started* covers installing Parallel Fortran on your machine and verifying that it is operating correctly.
- *Part II: Tutorial* introduces you to the operation of the compiler and the other tools supplied with Parallel Fortran. In particular, there are tutorial sections explaining parallelism on the transputer and the way in which this can be accessed from Parallel Fortran programs.
- *Part III: Language Reference* contains a complete specification of the language accepted by the Parallel Fortran compiler. This is ANSI Fortran 77, with certain extensions which are described in this part.
- *Part IV: General Reference* contains the detailed technical information which you will require to write sophisticated applications for the transputer using Parallel Fortran.
- The appendices at the end of this manual contain supplementary information in a condensed form, such as tables of compiler error messages.

Further Reading

Although this *User Guide* does include a complete description of Fortran 77, readers who are unfamiliar with this language are advised to consult one of the many introductory texts available.

In a similar way, the reader is assumed to be reasonably familiar with the operating system of the host computer being used. For personal computers made by IBM, this will usually be PC-DOS, which is supplied with a manual called *Disk Operating System Reference*[2].

For compatible machines made by other manufacturers, the operating system will usually be MS-DOS, described in *Microsoft MS-DOS User's Reference*[3]. These two operating systems are largely compatible, and their documentation is very similar. We will refer to "MS-DOS" in this manual to mean the operating system used on your machine. The term *DOS Reference Manual* will be used to refer to the appropriate manual.

References to these and other documents mentioned in this manual are collected in a bibliography, which can be found on page 541.

Conventions

Throughout this manual, text `printed in this typeface` represents direct verbatim communication with the computer: for example, pieces of Fortran text, commands to MS-DOS and responses from the computer.

In examples, text *printed in this typeface* is not to be used verbatim: it represents a class of items, one of which should be used. For example, this is the format of the Fortran `ASSIGN` statement:

```
ASSIGN label TO int
```

This means that the statement consists of:

1. The word '`ASSIGN`', typed exactly like that.

2. A *label*: not the word '*label*', but something which the accompanying description explains.

3. The word '`TO`', typed exactly like that.

4. An *int*: once again, the accompanying description explains what this is.

In examples, it is sometimes necessary to indicate exactly where there is a space, or how many spaces are present. In these cases, we represent a space by the symbol '␣'.

Part I

Getting Started

Chapter 1

Installing the Compiler

This chapter contains instructions on how to load Parallel Fortran from the supplied floppy disks onto a hard disk ready for use.

You can skip this chapter if the compiler has already been installed on the machine you are using.

The compiler is distributed on three 360KB floppy disks. The contents of these disks are described in detail in appendix A.

To install Parallel Fortran on your hard disk, follow this procedure.

1. Place the disk labelled `Disk 1 of 3` in your floppy disk drive `A:`.

2. Type the following commands:

```
C>a:

C>install
```

3. Answer any questions the `install` program asks you.

4. Place the appropriate disks in drive `A:` when the `install` program asks for them.

It is important to use the supplied `install` program to install Parallel Fortran. If you simply copy the files, the installation will not be performed correctly.

Parallel Fortran will be installed on directory `tf2v1`. If this directory already exists, files in it with the same name as Parallel Fortran files will be overwritten.

The compiler is now installed, but can only be run in the directory `\tf2v1`. Before the compiler can be used from other directories `\tf2v1` must be added to the MS-DOS *search path* . Program files stored in directories which are on the search path can be loaded and run simply by typing the name of the program as a command. So, to make sure that the Fortran compiler (`t4f` or `t8f`) is available as a command, `\tf2v1` must be added to the search path.

The search path for your machine is set up by the batch file `c:\autoexec.bat` which is automatically executed when the machine starts up. To change the path, you will need to edit the `autoexec.bat` file using a text editor like `edlin`. (The *DOS Reference Manual* explains how to use `edlin`). `autoexec.bat` will probably already contain a line of the following form:

```
path ... list of directories ...
```

For example:

```
path c:\dos;c:\utils
```

In this case, you will need to add the text "`;c:\tf2v1`" on to the end of the line, giving:

```
path c:\dos;c:\utils;c:\tf2v1
```

If there is no `path` line in the `autoexec.bat` file, just add the line:

```
path c:\tf2v1
```

Three important points about setting the search path should be noted:

1. The documentation for previous versions of 3L compilers, including Fortran, recommended the use of a **set path=** command to set up the search path. This is equivalent to the **path** command, and can be changed to include **\tf2v1** in the same way.

2. If you already have an earlier Fortran compiler installed on your machine, a directory such as **\tf2v0** will be in your path; it should be removed before adding **\tf2v1**.

3. If you are a user of the Inmos TDS environment, your search path will probably include a reference to the directory where the TDS is held, such as **\tds2dir**. This reference must not precede **\tf2v1** in the path; if it does, the wrong version of the **afserver** program will be called.

4. If you are a user of Parallel C or Parallel Pascal, you should be aware that this version of Parallel Fortran includes new versions of the **afserver** program and the linker. The versions of these which were released with version 2.1 and earlier of Parallel C and version 2.0 of Parallel Pascal should not be used with this version of Parallel Fortran. This means that directory **\tf2v1** should precede the installation directories for the other languages in your search path.

Once **autoexec.bat** has been changed, you will need to reboot your machine to make the changes effective.

When you have completed this installation procedure, the compiler may be accessed from any directory on the disk on which it has been installed, in this case **C:**. If you wish to execute the compiler from a directory on some other disk, say **D:**, you should refer to section 17.3 in part IV of this manual for information about file handling in INCLUDE statement statements.

Chapter 2

Confidence Testing

This chapter describes a short procedure which may be followed to check that installation has been done correctly.

1. Set the current disk to the same disk as the compiler has been installed on. For example, if the compiler has been installed in directory `c:\tf2v1`, do this:

```
D>c:

C>
```

2. Set the current directory to a convenient directory for doing this test. For example:

```
C>cd \mine

C>
```

   NB: Don't use directory `\tf2v1` for the confidence test, as this would mean that you would not be testing whether the correct search path has been set up.

3. Check that the correct versions of the **afserver** program and of the compiler are available, by typing the following command. You should see the output shown.

```
C>t4f /i -:i
IBM PC Filer server Inmos V1.3 (14th October 1987) / 3L
V1.3.5
Copyright INMOS Limited, 1985
Transputer Fortran 77 compiler, F77_transputer V2.1.1
Copyright (C) EPCL 1990
Copyright (C) 3L 1990

C>
```

   If the above message does not appear, check the installation procedure, and in particular, ensure that the correct **path** command has been set up.

   If, after the **afserver**'s identity, the computer outputs the following, or something similar—

```
Last command = 0
Server terminated: bad protocol when expecting INT32
```

   —it is likely that there has been some error in setting up the transputer board. In particular, please check that the wire links, accessible from the back of the PC, have been correctly installed. The transputer board's documentation should help with this.

4. Copy the example **hello.f77** file to the current directory:

```
C>copy \tf2v1\examples\hello.f77
     1 File(s) copied

C>
```

5. Compile the example using the T4 version of the compiler (this will work for the T8 as well, because the example contains no floating-point instructions):

```
C>t4f hello
```

```
C>
```

6. Link the resulting binary file with the necessary parts of the run-time library:

```
C>t4flink hello

C>linkt hello      \tf2v1\frtlt4 \tf2v1\t4harn

C>
```

7. Finally, the program can be run:

```
C>afserver -:b hello.b4
Hello, world!

C>
```

The output "`Hello, world!`" comes from the `hello.f77` example program. If it does not appear, we recommend that the installation procedure should be carefully repeated, and the confidence test procedure followed again. If this message still does not appear, please contact your dealer for further assistance.

Part II

Tutorial

Chapter 3

Developing Sequential Programs

This chapter shows you how to use the Parallel Fortran compiler to develop conventional sequential programs to run on the transputer. You should be familiar with the contents of this chapter before you progress to the later chapters explaining parallel programming on the transputer.

The instructions in this chapter assume that the Parallel Fortran compiler has already been installed as described in chapter 1.

Some of the operating procedures described here are different for T4 and T8 transputers. You should find out which type of transputer is fitted in your PC before using the compiler.

3.1 Editing

Any editor which handles standard MS-DOS text files can be used to create or change Parallel Fortran source programs. The example

below shows how the **edlin** editor supplied with MS-DOS can be used to create a new Parallel Fortran source program.

```
C>edlin hello.f77
New file
*i
      1:*       PROGRAM HELLO
      2:*       PRINT *,'Hello, world!'
      3:*       END
      4:*^C
*e

C>
```

The *DOS Reference Manual* explains how to use **edlin**.

Note that the "folded" files which the Inmos TDS works with are not ordinary MS-DOS text files and that therefore they cannot be used directly as input to the compiler. However, the **tdslist** utility program supplied with the TDS will convert TDS-format text files into ordinary MS-DOS text files which can be read by the Parallel Fortran compiler.

3.2 Compiling

A Parallel Fortran source program is compiled into a binary object (**.bin**) file of T4 transputer instructions by a command of the form:

t4f *source-file*

To compile code for a T8 transputer, use the command

t8f *source-file*

Note that, in general, code compiled for a T4 will not run on a T8 (or vice versa) so you must use the command appropriate for the type of processor in your transputer board.

The *source-file* is the filename of the Fortran source program which is to be compiled. If no filename extension is given in the command, `.f77` is added automatically.

So, to compile the file `hello.f77` for the T4, you would give the command

```
C>t4f hello
```

If the source file contains no errors, an output object file `hello.bin` is produced. If the compiler detects errors in the source program, it writes diagnostic messages to the MS-DOS standard output stream. Error messages may therefore be redirected using '>', or piped using '|'. The format of compiler error messages is described in section 17.6 in part IV of this manual, and a list of all the syntax error messages which the compiler may produce can be found in appendix G.

3.3 Linking

Once a Parallel Fortran program has been compiled into an object (`.bin`) file, it must be linked with any external subprograms it requires before it can be run, including intrinsic functions like `SQRT`, and other subprograms from the Parallel Fortran run-time library. This is done by the *linker*. Here we discuss the most usual linker operations; a full description of the linker can be found in chapter 19.

Rather than calling the linker directly, it is usually more convenient to use one of the batch files provided for the purpose.

To link T4 code produced by the `t4f` compiler use the command:

`t4flink` *object-file*

For example,

```
t4flink hello
```

To link T8 code produced by `t8f` use the command:

`t8flink` *object-file*

You must use the link command appropriate to the target processor (T4 or T8).

Both these batch files assume that the object file's extension is `.bin`, and produce an executable file with the same file name as the object file and extension `.b4`.

3.3.1 Linking More than One Object File

This section deals with linking more than one object file at a time. If you only want to link single object files for now, you can skip to section 3.4 which describes how to run executable files produced by the linker.

The `t4flink` and `t8flink` batch files can be used to link up to nine object files. As before, the extensions of all the object files are assumed to be `.bin`. The executable file generated will have the file name of the first object file specified, with the extension `.b4`.

For example, if there are two Fortran source files, `main.f77` and `fns.f77`, the following commands will compile them and link them together, producing an executable file for the T4 called `main.b4`.

```
C>t4f main

C>t4f fns

C>t4flink main fns
```

Compiling and linking the example files above for the T8 would be done as follows:

```
C>t8f main

C>t8f fns

C>t8flink main fns
```

3.3.2 Indirect Files

It is quite common for programs to consist of many different object files. The `t4flink` and `t8flink` batch files cannot handle more than nine, but even with fewer files than this, you may find the command line awkward to type.

The linker provides a way of getting round this problem, called an *indirect file*. An indirect file is a text file containing a list of object file names, all of which are to be included in the executable file. It is specified in the linker command by its file name preceded by an '**@**'. For example:

```
C>t4flink @objfiles
```

This will cause the linker to find the file **objfiles.dat**, and link together all the object files specified in it. As usual, the generated file will be given the name of the first object file with the extension `.b4`.

Indirect files are assumed to have the extension **.dat**. They contain a list of MS-DOS file names, with one file name on each line. Full path names, including directory specifications, are allowed. Indirect files may also include the names of other indirect files, by preceding with an '**@**'; nesting indirect files in this way may be done to five levels.

The example indirect file `objfiles.dat` above might contain the following text:

```
main
fns
\userlib\general\io
@grafpack
```

When used in the example given above, this will link the object files `main.bin` and `fns.bin` from the current directory and `io.bin` from the directory `\userlib\general`, together with all the object files specified in the indirect file **grafpack.dat**. The executable file generated will be `main.b4`.

3.3.3 Calling the Linker Directly

Occasionally, instead of using the batch files, you may need to call the linker directly, or write your own batch files to do so. Fuller information about the linker may be found in chapter 19, and details of the internal format of object files are provided in the Inmos *Standalone Compiler Implementation Manual*[13].

The linker is invoked by the command `linkt`. The general form of a link command is

```
linkt object-files,executable-file
```

object-files is a list of object file names separated by spaces. These are the object files which are to be linked together. All of them must have been compiled for the same processor type (T4 or T8). If an object file is specified without an extension, the extension is assumed to be `.bin`.

The order in which the object files are specified is significant. Details of this may be found in sections 3.5 and 17.2.4.5.

The *executable-file* is the name of the file to which the linker writes the executable output code. If no extension is specified, the linker supplies the extension `.b4`. The executable file and its preceding comma may be omitted; in this case, the executable file is given the same file name as the first object file in the command line, with the extension `.b4`. If the first file mentioned on the command line is an indirect file, the executable file is given a name taken from the name of the first object file listed in the indirect file.

To link Fortran programs, you must include in the list of object files both the Parallel Fortran run-time library and a special object file called a "harness". The directory `\tf2v1` contains two versions of both of these components: `frtlt4.bin` and `t4harn.bin` for T4 transputers, and `frtlt8.bin` and `t8harn.bin` for T8 transputers. The linker will not allow you to mix T4 and T8 object files.

The example below shows the command necessary to link all the files listed in the indirect file `subs.dat` into a single executable file for the T4, called `prog.b4`.

```
C>linkt @subs \tf2v1\frtlt4 \tf2v1\t4harn,prog
```

Note that the Parallel Fortran run-time library (`frtlt4.bin`) and the harness (`t4harn.bin`) must both be named explicitly as input object files.

For the T8, the command would be the following.

```
C>linkt @subs \tf2v1\frtlt8 \tf2v1\t8harn,prog
```

3.3.4 Libraries

It is often convenient to be able to treat a group of object files as a single unit. For example, the Parallel Fortran run-time library consists of many separate object files, but is supplied as a single file containing all of them.

The linker provides the option of linking together a group of object files to produce a *library* file instead of an executable file. The library contains all of the code and entry points defined by the input object files, which can be changed or deleted without affecting the library. To change a library it must be relinked from its component parts.

Library files have several advantages over using indirect files.

- The linker selects from the library file only those modules which are actually referenced elsewhere in the program; the others are not included in the executable file.
- Copying a single file to another place is simpler than copying many component object files and making sure that the corresponding indirect file is kept up to date with changes in directory and file names.

- Opening just one library file is faster than opening an indirect file and several object files.

However, using an indirect file may be faster while a library is being developed because there is no need to relink the library whenever a component module is changed.

A linker command of the form shown below is used to produce a library from a number of component object files.

```
linkt object-files,library-file/l
```

The option letter after the '/' is a lower case 'L'.

The form of the input *object-files* is the same as for normal operation of the linker: a list of filenames separated by spaces. Indirect files are indicated by an '@' sign as before.

The *library-file* must be a single MS-DOS file name. If no extension is specified, the linker will give it the extension `.lib`. Note that this is different from the default extension for input libraries, which is `.bin`.

The example below shows a graphics library being built from a core graphics module and two device driver modules. The library is then linked in the ordinary way with a user program. Indirect files are used to simplify the required linker commands.

```
C>type graflib.dat
core
tek
hp

C>linkt @graflib,graflib.bin/l

C>type myprog.dat
myprog
graflib
\tf2v1\frtlt8
\tf2v1\t8harn

C>linkt @myprog
```

3.4 Running

Executable programs are loaded into the transputer board and run using the `afserver` program, which runs on the IBM PC.

The `afserver` is an ordinary MS-DOS program, and after loading the Fortran program into the transputer board, it remains active throughout the program's run. Instructions are sent from the Fortran run-time library to the `afserver` whenever it needs to perform MS-DOS functions such as reading information from the disks, displaying output on the screen and so on. The results of these operations are sent by the `afserver` back to the transputer board.

The command to load and run a program is:

```
afserver -:b filename
```

The *filename* must be the name of an executable file produced by the linker. The file name extension must be specified. An example of a command to load and run a simple program would be:

```
C>afserver -:b hello.b4
```

Note that this will only work if your program uses a fairly small amount of stack memory. See section 3.5 for how to get round this problem.

Appendix section F.3 includes more information about the `afserver` and its options, and the Inmos *Stand Alone Compiler Implementation Manual*[13] (section 10) contains a full description. Note that the `-:e` (test error flag) switch described in [13] is not supported for use with Parallel Fortran programs. For improved performance, the Fortran compiler relies on being able to generate code which might incidentally cause the error flag to be set. Therefore, the transputer error flag may be set as part of the normal execution of a Fortran program.

The running of programs can be simplified by putting the appropriate `afserver` command into an MS-DOS batch file. Typing the

name of the batch file is then sufficient to run the program. For example:

```
C>type myprog.bat
afserver -:b \mydir\myprog.b4

C>myprog
```

The command **myprog** will then call **afserver** to load the executable file **\mydir\myprog.b4** into the transputer board and start it. Note that if a program compiled and linked for the T4 is loaded into a T8 (or vice versa) the effects will be unpredictable.

3.4.1 Using Fortran Programs as MS-DOS Commands

Because of the limitations on what can be done with MS-DOS batch files it is useful to have a way of running a transputer Fortran program as if it were an MS-DOS **.exe** file.

You can turn any **.b4** file into an MS-DOS command by making a copy of the file **\tf2v1\linkt.exe** in the same directory as the **.b4** file, giving it the same root filename as the **.b4** file but keeping the **.exe** extension. For example, if the current directory contains the executable file **ex.b4**, it can be run as a command by typing:

```
C>copy \tf2v1\linkt.exe ex.exe

C>ex
```

This new **ex** command can be used from any directory, provided the directory containing **ex.exe** and **ex.b4** is on the MS-DOS search path.

(**linkt.exe** works by taking the command verb from its command line, adding **.b4**, and then calling **afserver** to load that file from the same directory **linkt.exe** itself was loaded from).

When a `.b4` file is invoked via a "driver" program in this way, the `-:o 1` switch (see section 3.5) is added automatically and the program is given a large amount of stack space. If you want to run a program as an MS-DOS command, but with its stack in fast on-chip RAM, you should invoke the program as usual but add `-:o 0` to the command line (hyphen, colon, letter 'o', then a space followed by the digit zero). For example:

```
C>ex -:o 0
```

3.4.2 I/O Units, Redirection and Piping

Section 16.8.1.2 explains how a Fortran program's I/O units are preconnected. In particular, unit 5 is connected to the MS-DOS standard input facility and unit 6 to standard output.

Normally standard input comes from the keyboard, but it can be taken from a file by using the MS-DOS redirection symbol '<' in the normal way. For example, when running a program `wc`, you could redirect unit 5 to read from file `chap1.txt` by using this command:

```
C>afserver -:b wc.b4 <chap1.txt
```

This also works if `wc.b4` is invoked by a driver program, `wc.exe`:

```
C>wc <chap1.txt
```

Similarly, the standard output normally goes to the screen, and may be redirected using the '>' symbol. For example, `mangle.b4` is a program which reads raw data through unit 5, processes them and writes the results to unit 6. It could be made to read from file `raw.dat` and write to `cooked.dat` by this command:

```
C>afserver -:b mangle.b4 <raw.dat >cooked.dat
```

The output from unit 6 may also be *piped* into an MS-DOS *filter* program by writing the name of the filter after a vertical bar '|', as shown below.

```
C>afserver -:b mangle.b4 <raw.dat | more
```

The *DOS Reference Manual* describes in detail what can be done with filters. (The **more** program simply displays its input on the screen, a page at a time).

3.5 Memory Use

The memory used by a Fortran program is divided into four storage areas.

- *Code storage* is used to hold the executable instructions of the program itself, together with some constant data and control information.
- *Static storage* is used to hold the following:
  - arrays;
  - **CHARACTER** variables;
  - variables initialised by **DATA** statements, or by the extended forms of the explicit type statements;
  - variables in **COMMON** blocks;
  - variables which have been specified in **SAVE** statements.

  Static storage which is not initialised by the program is initialised automatically to zero.
- *Stack storage*(sometimes referred to as *workspace*) is used for all other variables, unless the program has been compiled using the **/S** switch, in which case all variables are held in static storage. The stack is also used for function calls and **CALL** statements and for holding pointers to the actual arguments of subprograms.

  In addition, library functions use varying amounts of stack space as working storage. The stack requirements of the mathematical functions are given in the Inmos *TDS Compiler Implementation Manual*[14] (Section 10, Parameters and workspace

requirements) and are generally about 40 to 100 words. The stack requirements of the floating-point arithmetic support library for the T4 are generally about 10 to 40 words. About 70 words of stack storage are permanently reserved for use by the run-time library.

Variables created on the stack are not automatically initialised to zero, and may have any initial value.

- *Heap storage* is used internally by the run-time library for I/O buffers, etc.

These four areas of storage are mapped onto two areas of physical memory:

- *On-chip* memory. The T4 has 2KB of fast on-chip memory, and the T8 has 4KB.
- *External* memory. The Inmos B004 board has either 1MB or 2MB of external memory.

Using the linker only, two methods of mapping the storage areas onto physical memory are available: the default method, and the alternative method. The configurers required for developing parallel programs give the user more advanced methods for controlling the use of memory. See sections 5.1.2 and 5.8, and chapter 26.

3.5.1 Default Memory Mapping

Default memory mapping is used if the **afserver** program is called as described in section 3.4 above. With this arrangement, the T4's on-chip memory, and the first 2KB of the T8's on-chip memory, are used for stack storage. Since on-chip memory is faster than external memory, programs can run much faster with default memory mapping. Obviously, you must be certain that the program's stack storage will fit in the available 2KB.

If you are using a T8, default memory mapping provides an opportunity for further speed inprovements, since the remaining 2KB of the T8's on-chip memory is available for code storage. To take advantage of this, you should place small, speed-critical subprograms at the beginning of the link-list.

WARNING: A program which exceeds the amount of available stack space will fail in unpredictable ways: it may hang, or it may simply give wrong answers.

3.5.2 Alternative Memory Mapping

Unless you are sure your program's stack data will fit into the 2KB of available on-chip memory, you should run it like this:

```
C>afserver -:b myprog.b4 -:o 1
```

The `-:o 1` switch (hyphen, colon, option letter 'o', then a space, then the digit one) changes the way memory is allocated to give the program a very large amount of stack space. In this mode of operation, the size of the stack is only limited by the amount of external RAM available, but execution speed is lower because the external RAM used for the stack is slower than the transputer's on-chip RAM.

3.5.3 Limit on Program Memory

The current version of the linker generates executable files which will only run correctly on boards having 1MB or 2MB of memory. To get round this restriction, the Parallel Fortran kit includes the `mempatch` program which may be used to change executable files to run on boards which have different amounts of memory. See chapter 20 for a discussion of `mempatch`.

3.6 Accessing MS-DOS Functions

The Parallel Fortran run-time library for the transputer includes a number of functions and subroutines which allow Fortran programs to access the MS-DOS host system. These functions are described in detail in section 18.2.2.

All MS-DOS functions are accessed by sending a set of register values to the host processor, executing a software interrupt instruction and finally receiving a set of modified register values. Thus, to use these functions you require a detailed knowledge of register uses and interrupt numbers for the MS-DOS function you wish to use. One source of this information is the IBM *DOS Technical Reference*[4].

Register values are moved to and from the host machine in an integer array known as a *DOS block*. The programmer is helped in building a DOS block by various constants defined in the file `DOS.INC`, and this should be included in any subprogram which makes use of MS-DOS functions. The length of the DOS block is the value of the constant `F77_DOS_BLOCK_SIZE`, defined in `DOS.INC`. The elements of the DOS block correspond to 16-bit registers of the MS-DOS machine, and to help with accessing these, `DOS.INC` defines various constants whose names use the familiar Intel register names. This means, for example, that to load the value 10 into the `AX` register for moving to the host, the programmer could code the following:

```
INCLUDE 'DOS.INC'
INTEGER DOSBLOCK(F77_DOS_BLOCK_SIZE)
:
DOSBLOCK(F77_DOS_AX) = 10
```

There are a number of significant points about the format of the DOS block.

1. The registers in the DOS block are in a different order from that conventionally used under MS-DOS. This difference simplifies the job of the run-time library, and should not be visible to pro-

grammers who make use of the constants defined in DOS.INC as described above.

2. The transputer has no 16-bit data types; the 16-bit 80*x*86 registers such as AX are therefore represented in the DOS block by Fortran INTEGER variables. Again, programs which access DOS blocks should not normally be aware of this difference.

3. The byte registers of the host machine (AH, AL etc) are mapped onto the 16-bit word registers in the usual way. This means that accessing these registers will often require the use of the bit-manipulation functions, described in appendix E. For example, to load 10 into register AH for moving to the host, the following code should be used:

```
INCLUDE 'DOS.INC'
INTEGER DOSBLOCK(F77_DOS_BLOCK_SIZE)
:
DOSBLOCK(F77_DOS_AX) =
1   IOR(IAND(DOSBLOCK(F77_DOS_AX),255),ISHFT(10,8))
```

An MS-DOS interrupt is performed by using the F77_HOST_INTERRUPT subroutine. One of its parameters is a DOS block, into which the programmer loads the required values for the host registers before the call; the returned values of registers may be found there too. The segment registers used on the host may be used unchanged, or the segment register values in the DOS block may be used, depending on the value of the ISEGS parameter. The subroutine F77_READ_SEGMENTS is provided to read the contents of the segment registers so that particular registers can be changed while leaving the rest with their current values.

Some of the more complex interrupt calls, both to MS-DOS and to add-on packages like GEM and MS-WINDOWS require parameters and data blocks to be passed in memory rather than in registers. If a block of memory is required as a parameter to an interrupt call, it must first be acquired from MS-DOS. After use, the memory block should be returned to MS-DOS so that it may be used again. These operations can be performed either by the appropri-

ate DOS function calls (described in the *DOS Technical Reference*) or by the run-time library subprograms `F77_ALLOC_HOST_MEM` and `F77_FREE_HOST_MEM`.

The Intel 80x86 architecture uses a 32-bit quantity to specify an address in memory. This can be held in a Fortran `INTEGER`, and consequently, for example, `F77_ALLOC_HOST_MEM` returns the address of the allocated memory as an integer. The more significant 16 bits of this object are a segment number and the least significant 16 bits are an offset from the base of that segment. The two parts of the address can be extracted using the bit-manipulation functions.

Because the transputer and its host MS-DOS system do not have any shared memory areas, information destined for a parameter block in the MS-DOS host cannot be simply written into the block by normal Fortran assignment operations. Instead, a duplicate of the block is created as a Fortran structure in the transputer's memory, and a subroutine is then called to move the contents of the block in the transputer's memory to its counterpart in the memory of the MS-DOS host. Similarly, reading information from a block in host memory involves transferring the block into an identical structure in the transputer's memory and then accessing the latter. These two operations are performed by the run-time library subroutines `F77_BLOCK_TO_HOST` and `F77_BLOCK_FROM_HOST`.

Note that some structures used as parameter blocks to MS-DOS, GEM, MS-WINDOWS and so forth include 16-bit fields. For example, GEM parameter blocks contain many 16-bit fields representing x and y co-ordinates. Because the transputer has no 16-bit data objects, there is no straightforward way of representing these, and they would in most cases need to be packed into integers, using the bit-manipulation functions.

Some simple examples of the ways in which the above functions might be used will now be given. First, the following program calls MS-DOS function 2_{16} (Character Output) to display the character '`A`' on the screen. The argument is passed in the `DL` register.

```
      PROGRAM SENDA
      INCLUDE 'DOS.INC'
      INTEGER DOSBLOCK(F77_DOS_BLOCK_SIZE)
C     Load 2 into AH and 'A' into DL
      DOSBLOCK(F77_DOS_AX) = ISHFT(2, 8)
      DOSBLOCK(F77_DOS_DX) = 65
C     Call interrupt 33 (hexadecimal 21)
      CALL F77_HOST_INTERRUPT(33, 0, DOSBLOCK)
      END
```

Next, a much more complicated example in which MS-DOS function 9_{16} (Output Character String) is used to print the string "Hello" on the screen. The string to be printed is written into a block of MS-DOS memory before the call.

```
      PROGRAM PHELLO
      INCLUDE 'DOS.INC'
      INTEGER DOSBLOCK(F77_DOS_BLOCK_SIZE), MEMORY
      CHARACTER*6 HELLO
      DATA HELLO/'Hello$'/
C
C      Allocate host storage and write string to it

      MEMORY = F77_ALLOC_HOST_MEM (6)
      CALL F77_BLOCK_TO_HOST (HELLO, MEMORY, 6)
C
C      Find current segment register addresses

      CALL F77_READ_SEGMENTS (DOSBLOCK)
C
C      Set up function call number
      DOSBLOCK (F77_DOS_AX) = ISHFT (9, 8)
C
C      Set up segment number and offset of string
      DOSBLOCK (F77_DOS_DX) = IBITS (MEMORY, 0, 16)
      DOSBLOCK (F77_DOS_DS) = IBITS (MEMORY, 16, 16)
C
C      Perform the call
      CALL F77_HOST_INTERRUPT (33, 1, DOSBLOCK)
C
C      Free the string memory in host
      CALL F77_FREE_HOST_MEM (MEMORY)
      END
```

Chapter 4

Introduction to Parallel Fortran

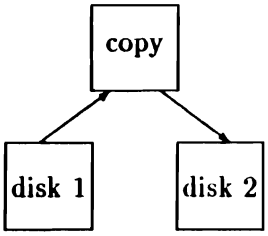
This chapter aims to help you become familiar with Parallel Fortran and its terminology. If you know occam, or if you have read a lot about the transputer, then you will already be familiar with the ideas on which Parallel Fortran is based. If not, don't worry; the ideas are quite simple. They are explained in outline here, and again in more detail in the chapters which follow.

4.1 Abstract Model

The treatment of parallel processing in transputer systems is based on the idea of *communicating sequential processes.* In this model, a computing system is a collection of concurrently active sequential processes which can *only* communicate with each other over *channels.* A channel connects exactly one process to exactly one other process. A channel can only carry messages in one direction: if communication in both directions between two processes is required, two channels must be used. Each process can have any number of

input and output channels, but note that the channels in a system are fixed; new channels cannot be created during its operation.

For example, a disk copy command built into a computer's operating system could be described as three concurrently executing processes: two floppy disk controller processes and one process doing the copying.



This example shows an important property of channel communications: they are *synchronised.* A process wanting to send a message over a channel is always forced to wait until the receiving process reads the message. In our example, this means that even if at some time the output floppy disk can't keep up with the input, the system will still work properly. This is because the copy process will automatically be forced to wait if it tries to send a message before the output disk process is ready to receive it. Sometimes it is useful to allow a sending process to run ahead of a receiving one; in such cases an explicit buffering process must be added to the system.

Note that because a process in this model is just a "black box" connected to the outside world only by its channels, the actual implementation of any individual process is not important. A process could be a bit of hardware or a software module; in particular it may also be another complex system, itself consisting of a number of communicating processes.

4.2 Hardware Realisation

The transputer was designed to be used as a component in concurrent systems of exactly the sort described in the previous section. Each transputer *processor* has four Inmos *links*, to connect it with other transputers. Each link has two channels, one in each direction. These hardware channels behave exactly like the abstract channels discussed above; they provide synchronised, unidirectional communication.

Arbitrary *networks* of transputers can be constructed simply by connecting their links together with ordinary *wires*, the only limitation being that each processor cannot be directly connected to more than four others.

At this level, a transputer can therefore be viewed as a single process in a multi-transputer system. However, it is also possible for any number of concurrent processes to be run on an individual transputer. Any word in the transputer's memory may be used as a channel to connect one internal process to another. The address of such a channel word is used to identify it to the transputer instructions (and Parallel Fortran subprograms) which send or receive messages. The contents of the word are used by the hardware to synchronise sending and receiving processes.

From a program's point of view, these internal channels and the hardware link channels are identical. The same instructions (or Parallel Fortran subprograms) are used to send and receive messages on both. Hardware link channels are identified by special fixed addresses. For example, on a T414 the input channel of processor link 3 is always at address $8000001C_{16}$. Internal channels have addresses allocated by software.

This equivalence of internal channels to hardware link channels means it is possible to develop a parallel system on a single transputer and then move some of its processes onto other transputers without having to recompile any code.

Each process executing on a transputer processor has a priority, which can either be "urgent" or "not urgent". The processor automatically shares its available time between these processes. A process can be *descheduled* either because it has performed an operation (such as sending a message to another process) which causes it to pause or, in the case of a "not urgent" process, because it has been executing without interruption for a certain period of time. The effect of this is that the CPU time-slices between the "not urgent" processes, but "urgent" processes are only interrupted when they cannot proceed because of a communication. For this reason, "urgent" processes should be designed so that they do not perform large amounts of computation, as they will "lock out" the less urgent processes entirely.

4.3 Software Model

Parallel Fortran is based on the same abstract model of communicating sequential processes as the transputer hardware.

A complete *application* is viewed as a collection of one or more concurrently executing *tasks*. Each task has its own region of memory for code and data, a vector of *input ports*, and a vector of *output ports*. The port vectors are known to the run-time library, and they can be accessed by special functions supplied with Parallel Fortran for the purpose. The code of a task is a single transputer image (`.b4`) file generated by the ordinary linker, `linkt`.

Tasks can be treated as software "black boxes" connected together via their ports, as shown in figure 4.1.

For example, a very simple task might accept on an input port a stream of `INTEGER` values each of which is an ASCII character, convert each character to upper case, and output the resulting stream of characters on an output port. The Fortran code for this is shown in figure 4.2.

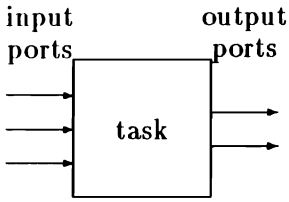


Figure 4.1: a task viewed as a "black box".

Tasks can be treated as atomic building blocks for parallel systems, to be wired together rather like electronic components. Indeed, several such basic building-block tasks are supplied with the compiler.

Ports are *bound* to real channel addresses by configuration software external to the task itself; the bindings can be changed without recompiling or relinking the task. Extended Fortran run-time library subprograms supplied with the compiler allow Fortran programs to send and receive messages over the channels bound to a task's ports. The port vectors themselves are not normally accesssible to a Fortran program, but the addresses of the channels to which they are bound may be found by use of the `F77_CHAN_IN_PORT` and `F77_CHAN_OUT_PORT` functions, as shown in figure 4.2.

The configuration software also provides ways of specifying which software tasks are to be run on which hardware processors. Each processor can support any number of tasks, limited only by available memory.

Tasks placed on the same processor can have any number of interconnecting channels. Tasks placed on different processors can only be connected where physical wires connect the processors' links. Each logical connection between two tasks placed on different processors is assigned exclusive use of one of the physical link channels connecting the processors. The number of interconnections between tasks on different processors is therefore limited by the number of hardware

```
      PROGRAM UPPER
      INCLUDE 'CHAN.INC'
      INTEGER INCHANADDR, OUTCHANADDR, WORD
      INCHANADDR = F77_CHAN_IN_PORT (0)
      OUTCHANADDR = F77_CHAN_OUT_PORT (0)
100   CONTINUE
         CALL F77_CHAN_IN_WORD (WORD, INCHANADDR)
         IF (WORD .EQ. -1) GOTO 200
         IF ((WORD .GE. 97) .AND. (WORD .LE. 122)) THEN
            WORD = WORD - 97 + 65
         END IF
         CALL F77_CHAN_OUT_WORD (WORD, OUTCHANADDR)
      GOTO 100
200   STOP
      END
```

Figure 4.2: Complete example task with one input and one output port.

links each one has. If more than four logical connections in each direction are required between one transputer and its neighbours, the designer of the system must provide explicit multiplexor tasks.

4.4 Multiple Input Channels

Sometimes, a task has to accept input from any of a number of input channels. For example, the main loop of a file server process would want to wait until a message was available from any one of its "client" processes. It can't read them all sequentially because a message could come from any one of them, in any order.

To handle this situation, the Parallel Fortran run-time library includes functions which enable the program to wait until one of a group of input channels is ready to transmit, and then report which channel it is. Alternatively, the program can "poll" a group of channels, that is, check which (if any) of them is ready to transmit, without waiting.

This facility corresponds roughly to that provided by the occam ALT statement, and is sometimes referred to as "guarded input".

4.5 Parallel Execution Threads

The software features described so far allow you to build parallel systems by connecting together the ports of a number of relatively independent tasks. In particular, all the tasks have separate code and data, and are only allowed to communicate with each other by sending messages over channels.

Parallel Fortran also allows you to take advantage of the transputer's architecture by creating new concurrent *threads* of execution within a task. Parallel Fortran's threads resemble the "processes" of Modula-2, and the "coroutines" of some other languages. Each one has its own stack (allocated by its creator), but shares its code with any other threads in the same task. Threads of the same task also have access to the same COMMON blocks, and *semaphore* functions in the run-time library are used to prevent threads which share data from interfering with each other.

Parallel Fortran's multiple thread facility differs somewhat from that supported by 3L Parallel C. It is a general rule that Fortran subprograms are not reentrant; that is, a subprogram may not be active more than once at any one time. This means that a subprogram cannot call itself, directly or indirectly; it also means that a subprogram may not be invoked through the thread mechanism more than once at a time. This precludes the easy use of the multiple threads for reading from a number of input channels "all at the same time". Nonetheless, the technique may be useful for almost any problem which falls into sections which are largely independent, and in particular many problems in simulation, real-time control and other areas map very well onto a multi-threaded algorithm.

4.6 Configuring an Application

Once an application has been designed and written as a collection of communicating tasks, how is it loaded into a physical network of transputers?

First, each individual task is built by compiling all its source files with the Fortran compiler and using the linker (`linkt`) to combine the resulting binary (`.bin`) files with the Parallel Fortran run-time library, and a special "task" harness in place of the standard harness we have used so far. This produces a task image (`.b4`) file.

Now a bootable application image file must be generated from the component task (`.b4`) files. The program which does this is called the *configurer*. It is driven by a user-supplied *configuration file* which specifies:

- the hardware configuration (processors, and the wires connecting them) on which the application is to be run;
- the names of the `.b4` files containing the component tasks of the application;
- the connections between the various tasks' ports;
- the placement of particular tasks onto particular processors in the physical network.

The output of the configurer is an application file which can be booted into the specified hardware network and run using the same `afserver` program used for simple standalone programs.

4.7 Processor Farms

The tools described so far allow you to build applications which execute on any transputer network whose wiring can be specified in

advance in a configuration file. For many parallel computations it is useful to be able to create applications which will automatically configure themselves to run on *any* network of transputers. Such applications will automatically run faster when more transputers are added to a network, without recompilation or reconfiguration.

Parallel Fortran allows you to create applications like this, provided the application can be implemented by a *processor farm*, and provided that there is enough memory on each processor in the network to support the required loading and message handling software.

In the processor farm technique, an application is coded as one *master* task which breaks the job down into small, independent pieces called *work packets* which are processed by any number of anonymous identical *worker* tasks. Work packets are automatically distributed across an arbitrary network of transputers by *routing* software supplied with the compiler. All of the worker tasks must run the same code. Each worker simply accepts work packets, processes them, and sends back *result packets* via the same routing software. A worker task is just a simple sequential loop: read a work packet; process it; send back a result packet; repeat.

Provided a master task can be written for your application which will split the job up into *independent* work packets which the worker tasks can handle without communicating with other tasks, you can use the *flood-fill configurer* to combine the code for the master and worker tasks into a bootable application file which can be loaded automatically into an arbitrary transputer network by the `afserver` program.

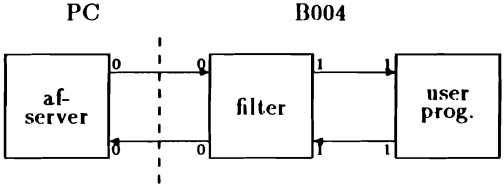
Many computationally intensive applications can in fact be implemented by processor farms, particularly graphics applications like ray-tracing where different sections of the screen can be worked on independently.

Chapter 5

Developing Parallel Programs

In this chapter we move on from looking at the general features of Parallel Fortran to explaining how some of the parallel programming tools supplied with the compiler are used in practice. The general-purpose configurer is described here along with the extended run-time library subprograms for sending messages over channels and creating new execution threads. Processor farm applications are covered in the next chapter.

We have actually already encountered an interesting example of a parallel system: even a simple sequential program running on a transputer board plugged into a PC runs in parallel with the `afserver` program on the host computer, as shown below.



The **afserver** task is an ordinary MS-DOS executable (**.exe**) file that runs on the PC. It loads executable **.b4** files into the transputer and also acts as a file server, handling I/O requests made by the transputer. The **afserver** and the transputer execute in parallel and communicate via an Inmos link. The messages sent to the **afserver** are normally generated by the Parallel Fortran run-time library. It converts I/O statements, for example **READ** and **WRITE**, into messages requesting the **afserver** to perform MS-DOS operations like *write 512 bytes* and then waits for the **afserver** to reply.

In principle, the **afserver** task could be directly connected to the user program. In practice, a *filter* task is interposed between them. The filter runs in parallel with the **afserver** and the user task; it simply passes on messages travelling in both directions. The filter is required because sometimes the messages passed between the user program and the **afserver** are only one byte long and the revision A T414 chip cannot handle single-byte message transfers on its hardware links. The filter pads out 1-byte messages to 2 bytes to avoid this problem.

5.1 Configuring One User Task

Up to now a standard "harness", **t4harn.bin**, has been linked in with all user programs. The harness contains system initialisation code, the filter, and a call to the user program. There is no need to describe the standard system configuration (**afserver**, filter and one user task) to the harness; the configuration is assumed.

Using the standard harness is simple but inflexible. We need a way to produce executable files for more complicated system configurations containing many tasks and many transputers. The configurer program supplied with the compiler can do this; a simpler harness (known as the "task harness") can then be used.

The configurer is driven by a user-written *configuration file* which describes the system to be built: the file lists all the physical proces-

```
!
! UPPER.CFG
!
processor host        !the PC
processor root        !the transputer in the B004
wire  jumper -        !connects...
      root[0] -       !link 0 of root transputer
      host[0]         !to the PC bus

task  upper    ins=2 outs=2            !the user task
task  filter   ins=2 outs=2 data=10k
task  afserver ins=1 outs=1

place afserver host  !afserver runs on PC
place upper root     !everything else on transputer
place filter root

connect ?   filter[0]   afserver[0]
connect ?   afserver[0] filter[0]
connect ?   filter[1]   upper[1]
connect ?   upper[1]    filter[1]
```

Figure 5.1: Configuration File with One Example Task

sors in the system, the wires connecting them, the tasks to be loaded into the system and their logical interconnections. The complete configuration file needed for a single transputer system with one task (i.e., the same configuration that is built into the standard harness) is shown in figure 5.1. In the rest of this section we will look at its contents in detail.

The example program we have chosen just converts a stream of characters read from the standard input stream to upper case. The Fortran source file, `upper.f77` is shown in figure 5.2 (the corresponding configuration file is called `upper.cfg`).

```
      PROGRAM UPPER
      CHARACTER*80 LINE
      INTEGER P, C
100   CONTINUE
         READ (5, 110, END=130) LINE
110      FORMAT (A80)
         DO 120 P = 1, 80
            C = ICHAR(LINE(P:P))
            IF ((C .GT. 96) .AND. (C .LT. 123)) C = C-32
            LINE(P:P) = CHAR(C)
120      CONTINUE
         WRITE (6, 110) LINE
      GOTO 100
130   STOP
      END
```

Figure 5.2: Fortran Source File for Upper Casing Program, `upper.f77`

5.1.1 Hardware Configuration

The first thing the configuration file needs to describe is the hardware configuration. A single B004 board plugged into a PC is very easy to describe.

```
processor host
processor root
wire jumper host[0] root[0]
```

There are two processors: the host PC and the root transputer in the B004. The root transputer is so called because if a larger network is built around a basic B004 system, the transputer directly connected to the PC becomes the root of the network—all communication with the file system on the PC must pass through it.

A wire connects the root transputer's link 0 to the host processor. The WIRE statement describes actual physical cables, in this case the little jumper you have to plug into the back of a B004 board which connects link 0 on the transputer to the PC bus.

Objects declared in the configuration language can have arbitrary names made up of letters, digits and the special characters '_' and '$', but are usually given mnemonic names. For example, a wire may be given a name, in this case `jumper`; or, more usually, it can be left anonymous, by writing '?' instead of the name.

The processor identifiers (`host` and `root`) used in a WIRE statement must have been declared in a previous PROCESSOR statement. This is a general rule: all objects in the configuration language (processors, wires, tasks) must be declared before they are used.

Now compare the short example above with the full configuration file in figure 5.1. You will notice a few differences in layout. Blank lines, spaces and tabs have been used to improve readability, and comments (from a '!' character to the end of the line) have been added. Some lines have been broken, indicated by a hyphen, '-', as the last non-whitespace character before a line break (or comment). Layout and comments are ignored by the configurer. Note that the configurer also ignores the case of letters: 'a' and 'A' are not distinguished.

As a short-cut when writing a configuration file, you can use the `worm` program to generate the hardware configuration automatically. This is done by using `worm`'s `/C` switch, and directing the output into a file. For example:

```
C>worm/c > upper.cfg
```

This would place a correct description of the current hardware configuration in the file `upper.cfg`. For a full description of the `worm` program, see chapter 22.

5.1.2 Software Configuration

As well as describing the hardware of a system, the configuration file must contain details of all its software tasks and their interconnections.

5.1.2.1 Declaring Tasks

For each concurrently executing task in the system the configuration file must contain a TASK statement which declares the number of input and output ports the task has. The **afserver** has only one input port (for file system requests) and one output port for responses.

```
task afserver ins=1 outs=1
```

Our example user task is next. It will be a program to convert characters to upper case, so it is given the name **upper**.

```
task upper ins=2 outs=2
```

As before, the **ins** and **outs** attributes specify the number of input and output ports for the task. The upper task has two of each, numbered from 0 upwards, and called `upper[0]` and `upper[1]`. Whether a port specifier like `upper[0]` refers to an input or an output port is determined by the context in which it is used.

The ordinary Parallel Fortran run-time library, with which the upper task will be linked, makes the assumption that the first two input and output ports of a task will be reserved for its use. The first pair of ports (numbered 0) have uses which will not be described here; they should simply be left unconnected. The second pair of ports (numbered 1) are assumed to be connected to a file server task. Here, we will connect the upper task to the **afserver** through a filter task.

The filter task has a slightly more complicated declaration:

```
task filter ins=2 outs=2 data=10k
```

The DATA attribute specifies the amount of memory a task needs. The `filter` task requires a minimum of 10K of workspace. For ready-made tasks supplied with the compiler, like `filter`, memory requirements can be looked up in the data sheets in chapter 28.

A user task like upper for which no memory requirement is specified gets all the free memory remaining once any other tasks placed on that processor are loaded. Only one task on each processor can have

its memory requirements left unspecified in this way. The configurer would otherwise have to decide how to split the remaining memory between several tasks with unspecified requirements. Because an even split is unlikely to be desirable in practice, this is not allowed. Section 5.8 below gives hints on estimating memory requirements in cases where multiple user-written tasks must be placed on the same processor.

5.1.2.2 Assigning Tasks to Processors

The placement of tasks on processors is specified by the PLACE statement. In our example, the **afserver** runs on the host PC and the user task (**upper**) runs on the root transputer with the filter task.

```
place afserver host
place upper root
place filter root
```

5.1.2.3 Making Connections between Tasks

The CONNECT statement establishes a channel between two tasks by binding the input and output ports. Because channels (unlike wires) are unidirectional, two CONNECT statements are needed to create channels going in both directions between the **afserver** and the filter.

```
connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
```

The CONNECT keyword can be followed by an identifier naming the connection, but all the configuration statements which declare new identifiers allow a question mark to be used in place of the identifier being declared. This is useful when there is no need to refer to an object after it has been declared. Currently there is no statement which can refer to the identifier declared by a CONNECT statement, so the question marks avoid the bother of naming essentially anonymous connections.

The first port mentioned in a CONNECT statement is an output port, and the second is an input port. In our example the connections are, in fact, symmetrical, but this is not always the case.

The remaining connections in our example system are written down in the same way:

```
connect ? filter[1] upper[1]
connect ? upper[1] filter[1]
```

5.1.3 Building the Application

Once a configuration file has been written all we have to do to execute the application is compile the Fortran source file **upper.f77**, link the resulting object file with the Parallel Fortran run-time library, and then run the configurer.

The example below shows what must be done to build an executable file from the uppercasing example:

```
C>t4f upper

C>t4ftask upper

C>config upper.cfg upper.app

C>afserver -:b upper.app
case changer
CASE CHANGER

^Z

C>
```

Two commands are new: **t4ftask** and **config**.

5.1.3.1 Linking for the Configurer

The ordinary batch file for linking Fortran programs (`t4flink`) is not suitable for linking a task because it links in the standard harness. `t4ftask.bat` is a batch file supplied with the compiler which links an object (`.bin`) file with the Parallel Fortran run-time library and a vestigial "task" harness containing neither the filter process nor any system initialisation code. The example below shows two Fortran source files, `main.f77` and `subs.f77` being compiled and then linked together to form a T4 task called `main.b4`.

```
C>t4f main

C>t4f subs

C>t4ftask main subs
```

Like `t4flink`, the `t4ftask` batch file can handle up to nine object files on the command line. If you need to link more files than this, you will need to use an indirect file, as described in section 3.3.2. Sometimes you may need to call the linker directly, as described in section 3.3.3; in this case, you must include the run-time library, `frtlt4.bin` and the vestigial *task harness*, `taskharn.t4`. Both can be found in the release directory, `\tf2v1`.

As usual, there are T8 versions of the batch file and the task harness. They are called `t8ftask` and `taskharn.t8`.

Note: it is important to link all tasks which are to be used with the configurer with the correct harness. If the wrong harness is used (for example by accidentally using `t4flink` *rather than* `t4ftask`*) then the configured application will fail to operate correctly. It may fail to execute, or it may simply give wrong answers.*

5.1.3.2 Running the Configurer

The configurer is invoked by the `config` command. Two filenames must be specified on the command line: first the configuration file,

then the name of the executable file to be output. For our case conversion example, the required `config` command line was:

```
C>config upper.cfg upper.app
```

The configurer does not supply default filename extensions, but `.cfg` is conventional for configuration files.

File names for the task images which make up the application are not supplied on the command line; the configurer derives them automatically by appending `.b4` to the task identifiers given in the configuration file. In our example, the configurer will search for task image files called `upper.b4` and `filter.b4`.

If a task image file is not found in the current directory, the configurer will automatically search all of the directories on the MS-DOS search path, so there is no need to make copies of ready-made tasks like `filter.b4` held in the same directory as the compiler (`\tf2v1`). The search path can be modified in the usual way by the MS-DOS commands `path` and `set`.

This automatic mechanism for specifying task image file names can be overridden by the FILE attribute of the configuration language's TASK statement, described in chapter 26.

Note that tasks placed on the `host` (PC) processor are not searched for in this way to be included in the output application file. The configurer does not attempt to load `afserver.b4` into the PC from the transputer! The `afserver` task must be declared and placed on the `host` simply in order to give a name to the object with which the `filter` task communicates over its port 0. As the `afserver` task loads the application into the transputer and handles all its I/O requests, it is reasonable to regard it as part of the configuration.

The output from the configurer can be run directly using the `afserver`:

```
C>afserver -:b upper.app
```

The actual configuration of the transputer network attached to your PC must match the declarations in the configuration file.

The memory requirements of configured tasks are specified in the configuration file; the `afserver` switches `-:o 1` and `-:o 0` are ignored by configured applications.

5.2 More than One User Task

In the previous section we saw how an application consisting of a single user task could be built using the configurer instead of the standard harness.

From this base, we can move on to more complicated systems containing multiple user tasks running in parallel.

Let's continue with the small case conversion example by splitting the job performed by `upper.f77` into two tasks: a driver task to handle file I/O, and a processing task which accepts a stream of words containing ASCII character code values on one of its input ports and sends the corresponding upper case character codes to one of its output ports.

This example is a bit contrived, but splitting a job up into an I/O task and a number of concurrent computation tasks is commonplace.

5.2.1 Inter-Task Communication Functions

Coding the driver task in Fortran is easy. Instead of checking for lower-case ASCII codes and subtracting 32 from them, it converts characters to upper case by sending a message containing the ASCII character code to the "computation" task and waiting for a reply message containing the result.

Parallel Fortran tasks send messages using the channel I/O functions described in chapter 18. The `CHAN` package provides functions to

```
      PROGRAM DRIVER
      INCLUDE 'CHAN.INC'
      CHARACTER*80 LINE
      INTEGER OUTCHANADDR, INCHANADDR, P, C
      OUTCHANADDR = F77_CHAN_OUT_PORT (2)
      INCHANADDR = F77_CHAN_IN_PORT (2)
100   CONTINUE
         READ (5, 110, END=130) LINE
110      FORMAT (A80)
         DO 120 P = 1, 80
            C = ICHAR(LINE(P:P))
            CALL F77_CHAN_OUT_WORD (C, OUTCHANADDR)
            CALL F77_CHAN_IN_WORD (C, INCHANADDR)
            LINE(P:P) = CHAR(C)
120      CONTINUE
         WRITE (6, 110) LINE
      GOTO 100
130   CALL F77_CHAN_OUT_WORD (-1, OUTCHANADDR)
      STOP
      END
```

Figure 5.3: `driver.f77` with Channel I/O Calls

send and receive messages of any length. The driver task is shown in figure 5.3; it uses `F77_CHAN_IN_WORD` and `F77_CHAN_OUT_WORD` to handle word-sized messages. A word is the same size as an `INTEGER`, 32 bits.

The driver source file, `driver.f77`, is included as an example in the distribution kit, along with the processing task, `upc.f77`, and a suitable configuration file, `upc.cfg`. These files can be found in the `examples` subdirectory of the directory containing the compiler, `\tf2v1`.

The statement in `driver.f77` which sends character codes to the processing task is:

```
CALL F77_CHAN_OUT_WORD (C, OUTCHANADDR)
```

The word (`INTEGER`) value to be sent is passed as the first argument in the function call.

Beware when using the channel I/O functions that sending and receiving tasks always agree on the size of messages. For example, if a task sends a word value as a single 4-byte message, the receiving task *must* read it as one 4-byte unit; it is not possible for the receiving task to read four separate 1-byte messages. Trying to do so may cause the transputer to lock up or behave unpredictably.

The second argument to `F77_CHAN_OUT_WORD` identifies the channel to which the message is to be sent. `OUTCHANADDR` has been initialised to the value of `F77_CHAN_OUT_PORT(2)`, that is, the address of the channel which is bound to output port 2. A CONNECT statement in the application's configuration file referring to `driver[2]` will specify which task the port is connected to. In our case, it will be the processing task to be described later.

The number of output ports a task has is defined by the OUTS attribute of the TASK statement used to declare the task in the configuration file. Our `driver` task has `outs=3`, so it has three output ports, numbered 0 to 2.

The value of OUTS is also accessible at run time, by using the function `F77_CHAN_OUT_PORTS`. It can be used to write tasks which handle an arbitrary number of ports, like the multiplexer task described later on in this chapter.

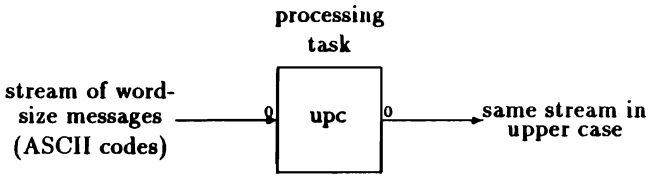
Using the functions `F77_CHAN_IN_PORTS` and `F77_CHAN_IN_PORT`, details of the task's input ports can be accessed in a similar way. In the driver example, the address of the channel bound to input port 2 is found by taking the value of `F77_CHAN_IN_PORTS(2)`.

The driver task reads records from unit 5 (that is, MS-DOS standard input), sends the characters one-by-one to the processing task, packs the reply messages (the translated characters) into records and writes these records to unit 6 (that is, MS-DOS standard output). It continues to do this until `READ` detects an end of input.

The next thing to look at is the processing task. It is logically a "black box" with one input port and one output port:

```
      PROGRAM UPC
      INCLUDE 'CHAN.INC'
      INTEGER OUTCHANADDR, INCHANADDR, C
      OUTCHANADDR = F77_CHAN_OUT_PORT (0)
      INCHANADDR = F77_CHAN_IN_PORT (0)
100   CONTINUE
         CALL F77_CHAN_IN_WORD (C, INCHANADDR)
         IF (C .LT. 0) GOTO 120
         IF ((C .GT. 96) .AND. (C .LT. 123)) C = C-32
         CALL F77_CHAN_OUT_WORD (C, OUTCHANADDR)
      GOTO 100
120   STOP
      END
```

Figure 5.4: The Processing Task



A Parallel Fortran implementation of this task is shown in figure 5.4.

The processing task uses the same channel I/O functions as the driver to send and receive messages. It terminates when it receives a -1 from the driver.

Extending the configuration file for our first, single-task, example (fig. 5.1) to handle two tasks is easy. We just change references to the old `upper` task to `driver`, and add the following extra configuration statements to describe the processing task and its connections.

```
task upc ins=1 outs=1 data=1k
place upc root
connect ? driver[2] upc[0]
connect ? upc[0] driver[2]
```

This says that the new task `upc` has one input port, one output port,

and requires 5KB of memory (section 5.8 gives hints on estimating task memory requirements). The upc task is placed on the root transputer, and its ports are connected to the corresponding ports of the driver task.

5.3 Building Multi-Task Systems

We will run into a problem when trying to compile and link the components of the dual-task system.

The ordinary Parallel Fortran run-time library expects to send messages to the **afserver** on output port 1 and receive replies on input port 1. This is true even if your Fortran program does not explicitly use any I/O statements—the library will still try to open the standard input and output streams and preconnect them to units 5 and 6.

This means that even though it does no Fortran I/O, the upc task will still attempt to communicate with the **afserver** if it is linked with the standard run-time library. However, the **afserver** is already connected to the driver task. The **afserver** task can't simply be shared between the driver and upc tasks, because that would require connecting one port on the **afserver** task to two client ports. That is not allowed—channels must always connect one port to exactly one other port.

This is not as restrictive as it seems, because a *standalone* version of the Parallel Fortran run-time library which does not need to communicate with the **afserver** is supplied with the compiler. The standalone library is just the same as the ordinary library except that all the functions which require **afserver** support (I/O, DOS calls, etc.) are missing.

A multi-task application must be split up into an I/O task with **afserver** support and one or more processing tasks which do not

need ordinary Fortran I/O because they use the channel I/O functions like **F77_CHAN_IN_WORD** instead.

Our example application is already in the right form: all we need to do is link the driver task with the standard run-time library and link the processing task, upc, with the standalone library.

In practice this logical organisation of an I/O task serving a number of parallel computing tasks is commonplace anyway. For embedded systems which do not need disk I/O support, all the component tasks may be linked with the standalone library, producing a consequent reduction in code size due to the absence of I/O initialisation code from the standalone library.

A batch file analogous to **t4ftask** is provided for linking an object file with the standalone library. It is called **t4fstask.bat**; a T8 version (**t8fstask**) is also supplied. As usual, these batch files can be used to link up to nine object files; if you need to drive the linker yourself, the files to link with are **safrtlt4.bin** and **taskharn.t4** in the release directory, **\tf2v1**, or their T8 equivalents. The commands required to link and configure the upper case example for a T4 are shown below.

```
C>t4f driver

C>t4ftask driver

C>t4f upc

C>t4fstask upc

C>config upc.cfg upc.app

C>afserver -:b upc.app
xyz123
XYZ123

pqr
PQR

^Z
```

You can try this out for yourself by making a copy of the relevant files, which are supplied in the directory **\tf2v1\examples**.

5.4 Multi-Transputer Systems

If you have followed the examples this far, the generalisation from a multi-task system running on a single transputer to a full multi-transputer system will be fairly obvious. All that is required is a change to the configuration file to describe the extra hardware and place some tasks onto processors other than the root transputer.

We could run the case conversion example on a two-transputer system with the driver task on the root transputer and the upc task on the other transputer. The extra hardware must be declared in the configuration file:

```
processor addon
wire ? root[1] addon[0]
```

This gives a name (addon) to the second processor and declares that it will be connected by a wire from its link 0 to link 1 on the root transputer. (Link 0 on the root transputer is already being used to connect it to the host computer).

If we reconfigured the application at this stage, the addon processor would be unused because the upc and **driver** tasks are both placed on the root transputer. We can fix this by modifying the PLACE statement for upc.

```
place upc addon
```

Now the configurer will automatically generate all the bootstrap and loader software required to make sure that the code of the upc task is loaded into the second transputer when the complete application is started on the root transputer by the **afserver**.

```
C>config upc.cfg upc.app
```

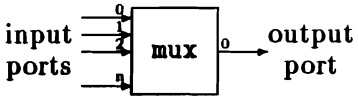
```
C>afserver -:b upc.app
two transputers...
TWO TRANSPUTERS...

^Z
```

Further generalisation to an arbitrary system should be clear: just declare more processors and wires in the configuration file, place tasks on the processors and connect them together.

5.5 Multi-Channel Input

One thing we have not yet seen how to do is to wait for a message from any one of a number of concurrently executing tasks. For example, a multiplexer task which accepted messages on any of an arbitrary number of input ports and passed them on through a single output port would be a useful building block. It might be used to allow a number of tasks to share a single hardware link.



A task connected to the output port of the **mux** task sees a sequential stream of messages, even though they are coming from any number of input tasks, in any order.

5.5.1 The ALT Functions

To implement the **mux** task we will need a way of handling a number of input ports "all at the same time".

In Parallel Fortran, this is done by using one of the ALT functions. The implementation of the multiplexer task shown in figure 5.5, for example, uses **F77_ALT_WAIT_VEC**. The second argument to this function is an array, each element of which is the address of a channel.

```
C       MUX.F77:   Message multiplexer task
C
        PROGRAM MUX
C
        INCLUDE 'ALT.INC'
        INCLUDE 'CHAN.INC'
C
        INTEGER INVEC(128), INPORTS, INPORT, LENGTH,
     1          OUTPORT, BUFF(256)
C
        OUTPORT = F77_CHAN_OUT_PORT (0)
C       Put addresses of all input channels in INVEC
        INPORTS = F77_CHAN_IN_PORTS ()
        DO 100 I = 1, INPORTS
           INVEC(I) = F77_CHAN_IN_PORT (I-1)
100     CONTINUE
C
200     CONTINUE
C
C          Wait for a channel to be ready
           INPORT = F77_ALT_WAIT_VEC (INPORTS, INVEC)
           INPORT = INVEC (INPORT)
C          Read message-length word, and body of message
           CALL F77_CHAN_IN_WORD (LENGTH, INPORT)
           CALL F77_CHAN_IN_MESSAGE (LENGTH, BUFF, INPORT)
C          Write the message to port 0
           CALL F77_CHAN_OUT_WORD (LENGTH, OUTPORT)
           CALL F77_CHAN_OUT_MESSAGE (LENGTH, BUFF, OUTPORT)
C
        GOTO 200
C
        END
```

Figure 5.5: Multiplexer Task Using ALT Functions

The first argument specifies how many channels there are. The function waits until one of these channels has data ready to read, then returns its place in the array. The program can now read the data from this channel in the usual way.

5.6 Multi-Threaded Tasks

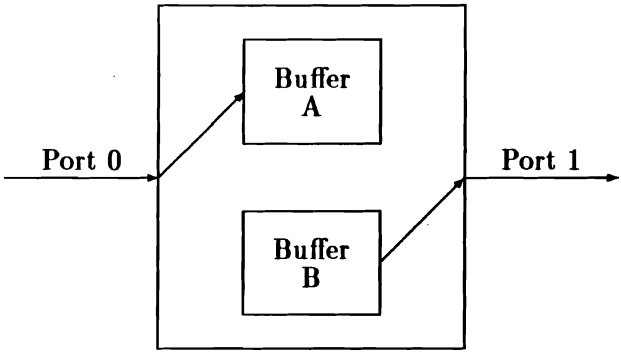
In this section we will look at an application of concurrent threads. This is a simple filter, the object of which is to copy 1024-byte blocks

of data from input port 0 to output port 1 as fast as possible. To do this, a *swinging buffer* technique is used. The task has two buffers, one of which is used for input and the other for output simultaneously. When both operations are complete, the buffers are swapped over. Figure 5.6 illustrates this arrangement.

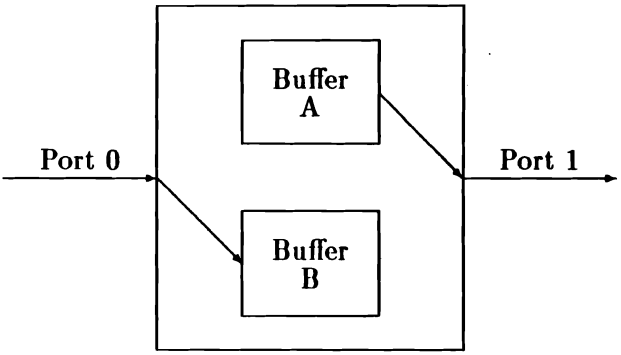
In order for the input and output to be performed simultaneously, the program creates a new *execution thread* to perform the input, while the main program does the output. The threads use two *semaphores* to coordinate their activities. This is necessary because the input thread needs to be sure that the contents of the buffer it is going to overwrite have already been output; while the main routine needs to be sure that a buffer is full of new data before it outputs it.

Figure 5.7 shows an implementation of this task in Parallel Fortran. The thread for performing the input is created by the call to `F77_THREAD_CREATE`, which invokes the `INPUT` subroutine in parallel with the execution of the main program, and passes it one argument, which is the address of the channel to use for input. Common blocks are shared between subprograms running in parallel in the usual way; in this case, the common block `BUFFS` is shared between the two threads. The semaphores `READY` and `INP` are part of this block, and so are accessible to both threads. The main thread signals that it is ready for more input by signalling the `READY` semaphore; the input thread signals that it has completed input by signalling the `INP` semaphore. There are two buffers; while one is being input to by the input thread, the other is being output from by the main thread. In fact, both the buffers are held in the array `BUFF`; one buffer starts at `BUFF(1)`, and the other at `BUFF(256)`. The input thread tells the main thread which buffer it has filled by placing its subscript in `P`.

If you haven't used semaphores or a similar method for controlling concurrent access to shared objects before, you should read a good introduction to the subject, such as [5,6], or restrict yourself to the stylised usage shown in the example. It is possible to introduce difficult-to-trace errors into a program if threads forget to synchro-



Input to Buffer A, Output from Buffer B



Input to Buffer B, Output from Buffer A

Figure 5.6: Filter task: Use of Swinging Buffers

```
      SUBROUTINE INPUT (CHANADDR)
      INTEGER CHANADDR
      INCLUDE 'CHAN.INC'
      INCLUDE 'SEMA.INC'
      COMMON /BUFFS/ BUFF(512), INBUF, OUTBUF,
     +               INP(F77_SEMA_SIZE), READY(F77_SEMA_SIZE)
      INTEGER BUFF, INBUF, OUTBUF, INP, READY
      INBUF = 1
100   CONTINUE
         CALL F77_SEMA_WAIT (READY)
         IF (INBUF .EQ. 1) THEN
            INBUF = 257
         ELSE
            INBUF = 1
         END IF
         CALL F77_CHAN_IN_MESSAGE (1024, BUFF(INBUF), CHANADDR)
         CALL F77_SEMA_SIGNAL (INP)
      GOTO 100
      END

      PROGRAM COPY
      INCLUDE 'CHAN.INC'
      INCLUDE 'SEMA.INC'
      INCLUDE 'THREAD.INC'
      COMMON /BUFFS/ BUFF(512), INBUF, OUTBUF,
     +               INP(F77_SEMA_SIZE), READY(F77_SEMA_SIZE)
      INTEGER BUFF, INBUF, OUTBUF, INP, READY
      EXTERNAL INPUT
      INTEGER OCHAN
      LOGICAL LOG
      OCHAN = F77_CHAN_OUT_PORT(0)
      CALL F77_SEMA_INIT (INP, 0)
      CALL F77_SEMA_INIT (READY, 0)
      LOG = F77_THREAD_CREATE (INPUT, 10000, 1, F77_CHAN_IN_PORT(0))
      CALL F77_SEMA_SIGNAL (READY)
100   CONTINUE
         CALL F77_SEMA_WAIT (INP)
         OUTBUF = INBUF
         CALL F77_SEMA_SIGNAL (READY)
         CALL F77_CHAN_OUT_MESSAGE (1024, BUFF(OUTBUF), OCHAN)
      GOTO 100
      END
```

Figure 5.7: Swinging Buffers: Example of Use of Threads and Semaphores

nize access to a shared object by waiting for a semaphore.

5.6.1 Threads versus Tasks

Threads can be useful in many situations. They are just "lightweight" processes, corresponding to processes in Modula-2 or the co-routines of some other languages.

Compared with tasks, threads are:

- "lightweight"—they share their code, heap, static and external data memory with all the other threads created by the same task;
- they can share data and may communicate either by using channels like tasks, or via shared memory;
- all the threads of a single task run on the same processor, allowing them to share memory.

Tasks on the other hand are more substantial than threads:

- they only communicate via channels;
- each task has its own code and data areas, separate from all other tasks; code, including run-time library functions, is not shared between tasks, even tasks placed on the same processor; this is so that...
- a task can be moved to a different processor simply by reconfiguration.

Two operations to be performed concurrently can be usefully performed by threads rather than tasks if all of the following conditions hold.

- They will never need to be run on distinct processors.

- The operations are closely coupled, i.e., they share a lot of common code. Code is automatically shared between threads, but each task has its own copy of all of its code, including library functions, so that if necessary it can later be moved to a different processor without requiring recompilation or relinking.

- The operations logically operate on shared data structures. This may be more efficiently performed directly by concurrent threads than by tasks copying the data back and forth as messages when they are modified.

5.7 Debugging

Parallel Fortran is compatible with the 3L interactive debugger, Tbug. This can handle multiple-task applications, and debugs multi-transputer applications by loading all the tasks on one transputer. You can trace the execution of the tasks at source level, and monitor the contents of variables. Breakpointing and single-stepping are provided.

Apart from using Tbug, what can be done when a parallel system locks up or fails to work properly? A sequential program could be attacked by inserting extra debugging output statements at strategic points in the code.

In a multi-task system this will in general only be easy to do to an I/O server task linked with the standard library and directly connected to the `afserver`. Unless you design debugging messages into the communication protocol used between the various tasks in your system you will not be able to get debugging output from a standalone task to a screen driving task. Even building debug message formats into the protocols used by the tasks in your system may not be enough if the fault lies in the failure of some intermediate task to transmit messages correctly.

However, it is possible to get output directly from a standalone task to an output device by using a second host computer and transputer board combination as a debugging tool. The second system can be attached to a suspect node of the system, in the same way as an oscilloscope can be used to debug an electronic system.

One way of doing this is to relink the suspect task with the standard run-time library (rather than the standalone library) and place it on the transputer attached to the second host computer. Ordinary **PRINT** statements can then be inserted in the code; the results will be output directly by the **afserver** in the second PC and displayed on its screen. The configuration statements required would be like this:

```
processor host
processor root
wire ? root[0] host[0]              !as before
processor extra_PC type=PC
processor extra_B004              !plugged into extra_PC
task extra_afserver ins=1 outs=1
wire ? extra_B004[0] extra_PC[0]
wire ? extra_B004[1] root[1]

place extra_afserver extra_PC
place suspect_task extra_B004

connect ? suspect_task[1] extra_afserver[0]
connect ? extra_afserver[0] suspect_task[1]
```

The main thing to notice here is the **type=PC** attribute given to the **extra_PC** processor. This tells the configurer not to try and bootstrap any tasks into that processor. (The **host** processor is just a special case for which **type=PC** is assumed). To make this configuration work, you must start the **afserver** on the extra PC using the **afserver** command *without* the **-:b** option before starting the system under test. If no **-:b** option is present on the command line, the **afserver** does not attempt to bootstrap the network it is attached to; it will simply accept file I/O request messages over its links.

It is also possible to use this debugging technique if you don't have

another host and transputer board combination but do have another PC with an Inmos link adapter card. Relink the suspect task with the full run-time library rather than the standalone library, then reconfigure the system with input and output ports 1 of the task being debugged connected to the PC with the link adapter, as follows:

```
processor second_PC type=pc
task second_afserver ins=1 outs=2
place second_afserver second_PC

processor any_processor          !of network being debugged
wire ? any_processor[3] second_PC[0]

task suspect_task ins=2 outs=2  !connect [1]'s to afserver
place suspect_task any_processor
connect ? suspect_task[1] second_afserver[0]
connect ? second_afserver[0] suspect_task[1]
```

This technique has two advantages: it only requires an extra PC and link adapter card, rather than an extra PC and transputer board, and there is no need to change the placement of the suspect task.

A third technique uses the three spare links on a transputer board plugged into the extra PC to accept debugging messages from up to four separate tasks anywhere in the network being debugged and multiplex them onto its PC screen.

5.8 Estimating Memory Requirements

Section 3.5 has already discussed the various categories of data storage. As noted there, the data requirement for a task is the sum of the number of bytes required for static, stack and heap storage in all its modules.

The **decode** utility (see chapter 21) can be used to determine a module's static data requirement. **decode** displays the number of *words* (not bytes) of static data required by a module near the top of the output listing it produces, after the keyword STATIC. The

whole task also has one word of static space permanently allocated to each module.

Stack and heap requirements are more difficult to estimate; you must decide how much space to leave for all the subprograms which may be active at once, based on the sizes of individual data items. Each level of subprogram calling uses about five words of stack space in addition to the space required for variable data.

The heap is used internally by the run-time library to allocate storage for I/O buffers, and to supply a workspace for the `F77_THREAD_CREATE` function. Heap storage is currently allocated by the run-time library in blocks of 4KB, so if your task uses the heap be sure to allocate at least that much space for it.

In fact, static storage and the heap are allocated from a single memory area, from which static storage is taken first. What is left is then available for the heap, if needed. For further details, see chapter 26.

In addition to the amount of space you estimate your task actually needs, it is a good idea to leave at least 1 or 2KB of extra overflow space, unless you are absolutely sure the task will never require more space than you have calculated.

Bear in mind that if a task exceeds its stated memory requirements the whole system will probably crash, so err on the side of caution. A good rule of thumb would be to allocate at least 1KB to simple tasks which don't use the heap, and 8–10KB for tasks which do use the heap.

If the stack space required by a task is small enough it can be allocated from the transputer's on-chip RAM. The space available there is 2KB on a T414, 4KB on a T800 (the restriction to 2KB for the T800 does not apply for configured tasks). Placing a computationally intensive task's stack in fast on-chip RAM can produce dramatic speed improvements. The configuration language contains various attributes for the TASK statement which allow control over memory layout. These more advanced topics are covered in chapter 26.

Chapter 6

Global Input/Output

In the last chapter, we looked at how to build configured applications with more than one user task, whether running on one or more transputers. In this chapter, we shall see how to arrange for all these tasks to use standard Fortran I/O statements and other facilities which need the support of the `afserver` program.

6.1 One Transputer

We saw in section 5.3 that only one task can communicate with the `afserver`, and that this task was the only one to be linked with the full Fortran run-time library. All the other tasks were linked with the standalone library, and this precluded them from doing standard Fortran I/O, DOS calls and so on. Figure 6.1 shows, for example, a simple two-task application, and figure 6.2 shows the corresponding configuration file.

The problem is that the server only has one possible connection to one filter task, and the filter task has only one possible connection to a user task. We can get round this problem by placing a special multiplexer task between the user tasks and the filter tasks. This

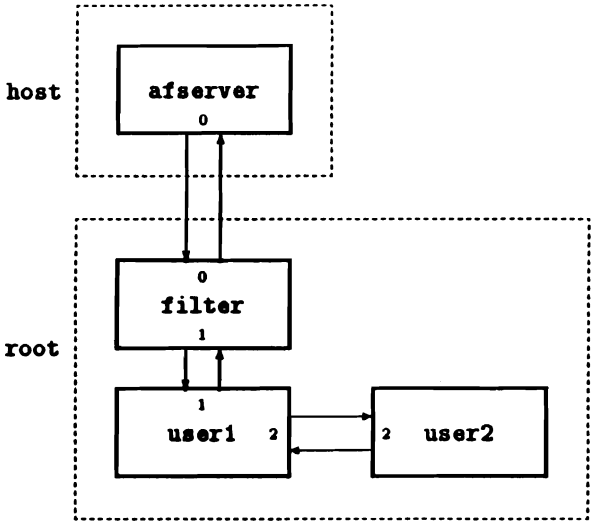


Figure 6.1: Two-task Application

multiplexer task is included with the Parallel Fortran kit, and is called **filemux**; a task data sheet for it can be found in chapter 28.

Figure 6.3 shows this arrangement. The configuration file is unchanged, except that the following statements, which connected user1 to the filter are removed:

```
connect ? filter[1] user1[1]
connect ? user1[1] filter[1]
```

and instead we have the following:

```
task filemux ins=3 outs=3 data=6656
place filemux root
connect ? filter[1] filemux[0]
connect ? filemux[0] filter[1]
connect ? filemux[1] user1[1]
connect ? user1[1] filemux[1]
connect ? filemux[2] user2[1]
connect ? user2[1] filemux[2]
```

```
processor host
processor root
wire ? root[0] host[0]

task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task user1 ins=3 outs=3 data=50K
task user2 ins=3 outs=3 data=50K
place afserver host
place filter root
place user1 root
place user2 root
connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? filter[1] user1[1]
connect ? user1[1] filter[1]
connect ? user1[2] user2[2]
connect ? user2[2] user1[2]
```

Figure 6.2: Two-task Application

Now it is **filemux** which is connected to the filter, and the two user tasks each have their number 1 port pairs connected to a **filemux** port pair. Each user task should be linked with the full run-time library using **t4ftask** or **t8ftask**, and each task can behave as if it has sole use of the **afserver**. The multiplexer arranges for all the messages from the user tasks to be transported to the **afserver** on the host, and transports the replies back to the correct user task.

You can arrange for the multiplexer to handle more tasks. Each must have their port pair 1 connected to a multiplexer port pair, starting at number 1 and going upwards with no gaps. For example, if the multiplexer is supporting 9 tasks, they must be connected to port pairs 1 to 9. The amount of memory which the multiplexer uses is no more than $(6 + 0.25n)$K bytes, where n is the number of tasks supported. So in the case of 9 supported tasks, the TASK statement should read:

```
task filemux ins=10 outs=10 data=7.75K
```

The multiplexer adjusts its own activities to support all the tasks

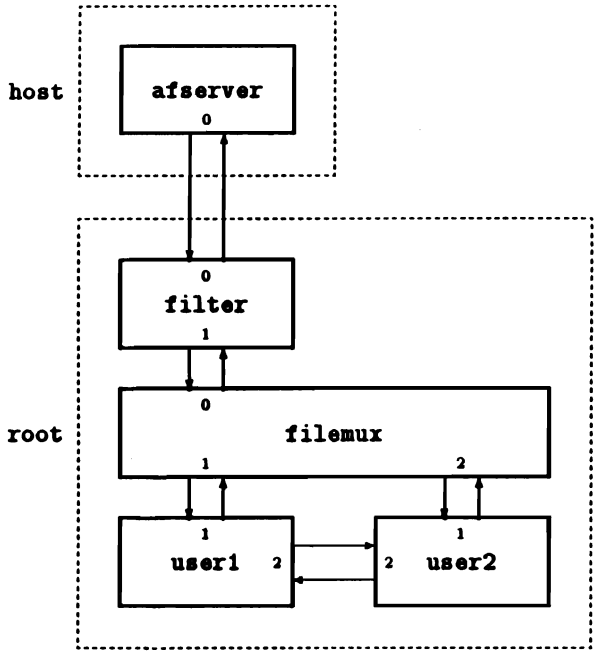


Figure 6.3: Two-task Application with Global I/O

which are connected in this way.

6.2 More than One Transputer

A task does not have to be on the same transputer as the multiplexer which supports it. Provided the necessary wires exist, it can be on an adjacent transputer. Figure 6.4 shows how this would be arranged, and figure 6.5 is the corresponding configuration file.

Each WIRE statement corresponds to a hardware link between two transputers, and supports two CONNECT statements, one in each direction. This means that the connections between `filemux` and one

supported task on a neighbouring transputer will use up one WIRE statement, that is, one hardware link. This implies two restrictions:

- If you have a task on a neighbouring transputer supported by a multiplexer on this, and you also want user tasks on the two transputers to be connected, you will need *two* hardware links between the two transputers.
- As a transputer has only four hardware links, the number of tasks on neighbouring transputers which can be supported is limited.

6.3 More than One Multiplexer

Fortunately, there is a way to improve on this situation. This can be done by using more than one copy of the `filemux` task.

Up to now, the number 0 port pair of the multiplexer has always been connected to the number 1 port pair of the filter task. However, it is also possible to connect the number 0 port pair to another copy of the multiplexer, which could be on another transputer. In this way, copies of the multiplexer can be built up into a tree. Figure 6.6 shows how this could be done, and figure 6.7 shows the corresponding configuration file.

Once again, a user task which is connected to the multiplexer, no matter how deep into the tree it is, can use the server's facilities as if it were directly connected. The task's server requests are passed up the tree of multiplexer tasks until they reach the `afserver`, and the response is similarly passed back to the correct user task.

6.4 Limits

The number of MS-DOS files and devices which the `afserver` can handle at the same time is limited, currently to 20. This means that

the network of tasks which are supported by **filemux** may not open more than 20 files at any one time. This applies regardless of the number of **filemux** tasks involved.

Each Fortran task which is linked with the full run-time library uses up two of this allotment of 20, for the pre-connected units 5 and 6. As a result, the maximum number of tasks which can be supported by the multiplexer network is currently 10.

6.5 Termination of an Application

When a task which is linked to the full run-time library terminates, for example by executing a **STOP** statement or calling the **EXIT** subroutine, it sends to the **afserver** a *server terminate* request. This causes the **afserver** to stop executing and return control to DOS.

Obviously, when a number of tasks are using the server, this cannot be allowed to happen. Accordingly, **filemux** does not pass on a server terminate request until all the the tasks it supports have tried to send one.

The effect of this is that the **afserver** does not terminate until it has been asked to do so by every task in the application which is supported by **filemux**. It is not enough for a task to go into a loop, or to be waiting for input; if this happens, the application as a whole will not terminate. Every task must terminate properly.

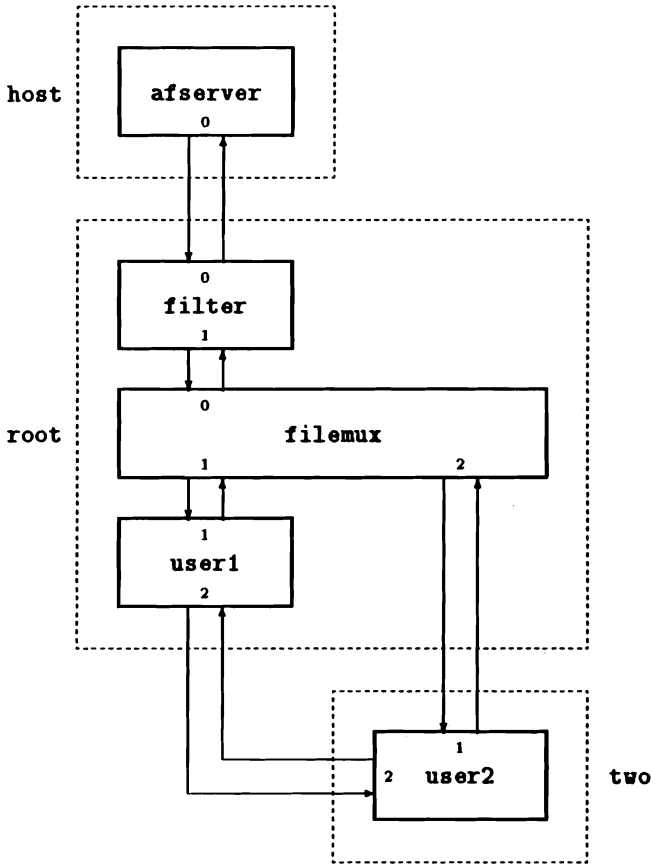


Figure 6.4: Task on Neighbouring Transputer

```
processor host
processor root
processor two
wire ? root[0] host[0]
wire ? root[1] two[0]
wire ? root[2] two[1]

task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task filemux ins=3 outs=3 data=6656
task user1 ins=3 outs=3 data=50K
task user2 ins=3 outs=3 data=50K
place afserver host
place filter root
place filemux root
place user1 root
place user2 two
connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? filter[1] filemux[0]
connect ? filemux[0] filter[1]
connect ? filemux[1] user1[1]
connect ? user1[1] filemux[1]
connect ? filemux[2] user2[1]
connect ? user2[1] filemux[2]
connect ? user1[2] user2[2]
connect ? user2[2] user1[2]
```

Figure 6.5: Task on Neighbouring Transputer

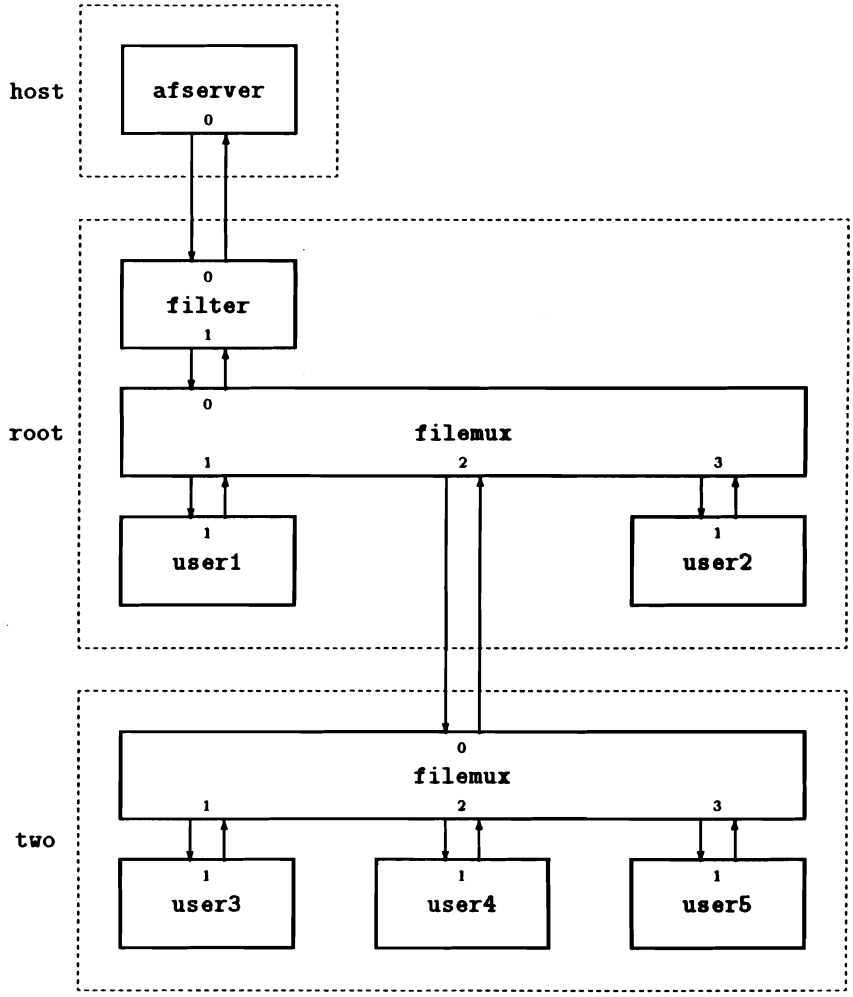


Figure 6.6: Networking Multiplexers

```
processor host
processor root
processor two
wire ? root[0] host[0]
wire ? root[1] two[0]

task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task mux1 file=filemux ins=4 outs=4 data=6912
task mux2 file=filemux ins=4 outs=4 data=6912
task user1 ins=2 outs=2 data=50K
task user2 ins=2 outs=2 data=50K
task user3 ins=2 outs=2 data=50K
task user4 ins=2 outs=2 data=50K
task user5 ins=2 outs=2 data=50K
place afserver host
place filter root
place mux1 root
place mux2 two
place user1 root
place user2 root
place user3 two
place user4 two
place user5 two
connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? filter[1] mux1[0]
connect ? mux1[0] filter[1]
connect ? mux1[1] user1[1]
connect ? user1[1] mux1[1]
connect ? mux1[3] user2[1]
connect ? user2[1] mux1[3]
connect ? mux1[2] mux2[0]
connect ? mux2[0] mux1[2]
connect ? mux2[1] user3[1]
connect ? user3[1] mux2[1]
connect ? mux2[2] user4[1]
connect ? user4[1] mux2[2]
connect ? mux2[3] user5[1]
connect ? user5[1] mux2[3]
```

Figure 6.7: Networking Multiplexers

Chapter 7

Processor Farms

The previous chapters showed how to create a parallel application for a multi-transputer system with a fixed hardware configuration. In this chapter we look at how to build one of the "processor farm" applications mentioned in the *Introduction to Parallel Fortran* in chapter 4 which will automatically flood-fill an arbitrary network of transputers with copies of a "worker" task.

Three things must be written to create a processor farm application:

1. A master task to split up the job into independent work packets.

2. A worker task, which is automatically copied to each node of the network.

3. A configuration file, describing the memory requirements and other attributes of the tasks.

In this chapter we will look at an example of a processor farm application. This is a program which displays pictures of the now-famous "Mandelbrot Set" on an IBM PC-type host equipped with a CGA-compatible display.

The full source code of the Mandlebrot master and worker tasks, and of the configuration file required, is printed in appendix J. These files are also supplied in machine-readable form in the `\tf2v1\examples` directory, along with a batch file (`mandel.bat`) which compiles, links and configures the example files into an executable application. Section 7.5 at the end of this chapter explains how to run the demonstration if you want to try it out before reading further.

The Mandelbrot program is suitable for running on a processor farm because each part of the final picture can be computed independently of all the others.

The master task has to split the job up into lots of small units which can be handled independently by the "farm workers". In the Mandelbrot case this is easy: the master divides up the screen area into 100 small squares, and sends the coordinates of the individual squares out into the network as work packets. Any idle worker receiving a packet calculates the required graphics display bitmap for that part of the picture and sends it back as a result packet.

7.1 The Worker Task

If you look at the code of the Mandelbrot worker task you will see that it is purely sequential. It consists of a single loop:

1. Get a work packet by calling `F77_NET_RECEIVE`. The work packet identifies the individual square of the display which is to be computed.

2. Work out the Mandelbrot values for that square, and place them in the `R_COUNTS` character variable in the results packet.

3. Send the result packet back to the master task by calling `F77_NET_SEND`.

4. Go back to step 1.

The F77_NET_SEND and F77_NET_RECEIVE functions are discussed below in section 7.3. Full details may be found in section 18.2.7.

The worker task does not care which processor it is executed on and must not communicate explicitly with other tasks. All communication between workers and master is handled "behind the scenes" by F77_NET_SEND and F77_NET_RECEIVE.

The only other restriction on the worker task is that because it must be replicated throughout the network and therefore cannot be directly connected to the **afserver** it must be linked with the standalone run-time library.

7.2 The Master Task

The master task of a processor farm application has three basic functions.

1. Split up the job into work packets. It sends the work packets out into the farm of worker tasks by calling F77_NET_SEND. The master simply does this as fast as it can: whenever the network of worker tasks becomes saturated, F77_NET_SEND is automatically blocked until a worker task becomes idle.

2. Receive result packets from the network by calling F77_NET_RECEIVE. If no result packets are available, F77_NET_RECEIVE will wait for one to arrive before returning.

3. Perform any I/O required by the worker tasks.

To prevent incoming result packets being blocked by the F77_NET_SEND function waiting for a worker to become free, or conversely the sending of work packets being blocked by F77_NET_RECEIVE waiting for a reply, these functions must be performed in parallel.

In the example implementation of the Mandelbrot program these functions are performed by three parallel execution threads: **SEND**, **RECEIVE** and **MAIN**.

7.3 The NET Package

A full description of the **NET** package subroutines may be found in section 18.2.7. Subprograms which call these subroutines should include the **NET** *package file*, by coding this statement:

```
INCLUDE 'NET.INC'
```

The administration of a processor farm is under the control of a task called **frouter** (see chapter 28). Each node in a processor farm contains a copy of this task; all the copies, and the master and worker tasks, are connected together by the flood-filling configurer (see section 7.4 below). This network of **frouter** tasks can be regarded by the programmer as a single entity, whose job it is to ensure that messages arrive at their correct destinations.

7.3.1 F77_NET_SEND and F77_NET_RECEIVE

F77_NET_SEND is used to send a message to the network, and F77_NET_RECEIVE is used to receive one from the network.

Messages sent to the network by the master task (using F77_NET_SEND) are routed to an idle worker task, if necessary passing through more than one node in order to reach one. At each level of re-direction, the messages are buffered. Only if all the worker tasks are busy, and all the buffering is full, will a call on F77_NET_SEND by the master task have to wait.

Messages sent to the network by worker tasks are routed back to the master task, once again passing through more than one transputer if necessary.

There is a limit on the size of a buffer that can be submitted to F77_NET_SEND; the constant F77_NET_MAX_PACKET_LENGTH is defined in the package file to have this value (currently 1024). If the message you wish to send is longer than this, it must be broken into a number of *packets*. The last packet of the message should be sent with the COMPLETE argument of F77_NET_SEND set to .TRUE.; this should also be done if there is only one packet in the message. All the other packets should be sent with COMPLETE set to .FALSE.. When a packet is received, F77_NET_RECEIVE sets its COMPLETE argument to the value used when the packet was sent. The network will ensure that a sequence of packets will arrive in the right order, but it is the receiving task's responsibility to fit the sequence of packets back together again.

It is best, however, to design the application to use messages which are smaller than 1024 bytes, as long packets can clog up the network and block packets being delivered to other nodes.

7.3.2 F77_NET_BROADCAST

Sometimes you may wish to start a run of your processor farm application by initialising all the worker tasks with the same set of data. These could be parameters obtained from the user, for example, or data tables which vary from run to run. This can be done using the F77_NET_BROADCAST subroutine.

F77_NET_BROADCAST should only be used by the master task. Each call on this subroutine results in a copy of the broadcast message being sent to every worker task in the processor farm. The broadcast message can be received by the worker tasks by using F77_NET_RECEIVE in the normal way. The most usual time to do a broadcast would be at the beginning of the run, but a message can be broadcast whenever the network is idle; that is, when all the work packets sent out by the master task have been answered by the worker tasks by sending a results packet. However, as there is no method to tell a broadcast message from a normal work packet, it

is up to the programmer to ensure that the worker tasks never get confused.

A broadcast message can be any length. If necessary, **F77_NET_BROADCAST** will break it up into packets for transmission through the network. In this case, the worker tasks will have to call **F77_NET_RECEIVE** more than once to receive it, checking the **COMPLETE** argument as described above.

Note that **F77_NET_BROADCAST** is the only reliable method to send an identical message to every worker task. Repeatedly calling **F77_NET_SEND** is unlikely to work.

7.4 Building the Application

Once the master and worker tasks have been compiled, the master should be linked with the standard run-time library (**t4ftask** for the T4 or **t8ftask** for the T8); the worker task must be linked with the standalone run-time library (**t4fstask** for the T4 or **t8fstask** for the T8).

The executable file containing the code of these tasks along with the extra software to flood-fill a transputer network with copies of the worker task is generated by the flood-fill configurer, **fconfig**.

7.4.1 Configuration File

Like the fixed-network configurer, **fconfig** requires a configuration file as input. This must specify at least:

- the filename of the master task;
- the filename of the worker task;
- the memory requirements of the worker task.

The configuration language accepted by **fconfig** is a subset of that accepted by **config**.

The minimum configuration file for the Mandelbrot example would be:

```
task master
task worker data=10k
```

fconfig would search for the master task in **master.b4**, and for the worker task in **worker.b4**. These file names can be over-ridden using the FILE attribute of the TASK statement, as shown below, but the task identifiers **master** and **worker** are special: you must use these names to identify the master and worker tasks to the flood-configurer.

If the alternative configuration file below were used, the configurer would expect to find the tasks in files called **mandelm.b4** and **mandelw.b4**.

```
task master file=mandelm
task worker file=mandelw data=10k
```

The DATA size specification is required for at least one of the tasks. Other attributes governing placement of stack memory in on-chip RAM and so on are covered in the reference part of this manual.

It is not required (and indeed not possible) to specify INS or OUTS for the master and worker tasks: all the ports and connections required are generated automatically by the configurer.

To run the flood-configurer, use a command of the form:

fconfig *configuration-file executable-file*

For example:

```
C>fconfig mandel.cfg mandel.app
```

The executable file generated by the flood-configurer will place the master task and one copy of the worker task on the root transputer, and distribute copies of the worker task to any other transputers

connected to the root. A filter task allowing the master task to communicate with the **afserver** is automatically added by `fconfig`, along with the loader and router tasks required to copy the workers across the network and carry messages between them and the master task.

This additional software occupies about 20KB of RAM in the current version of Parallel Fortran, so each node in our example network must have at least 32KB of RAM to support the 10KB worker task declared in the configuration file along with a router and loader. The root node must be larger again in order to support the master task as well.

7.5 Running the Example

A batch file, `mandel.bat`, is supplied along with the Mandelbrot example which will automatically compile, link and configure the application.

To run the program in a temporary directory, you can use the following commands:

```
C>cd \

C>mkdir temp

C>cd temp

C>copy \tf2v1\examples\*.*

C>mandel
:
```

The resulting executable file (called `fmandel.b4`) can be loaded and run on any network containing only T4 transputers. To use T8 transputers you would have to recompile the tasks to generate T8 code. Section 7.6 below describes how to flood-configure applications to run on a network containing a mixture of T4 and T8 processors.

The executable file can be loaded and run in the normal way:

```
C>afserver -:b fmandel.b4
```

When it starts, the Mandelbrot program reminds you that it needs an IBM PC compatible host machine with CGA graphics to work properly, then prompts you to enter several numeric parameter values on the keyboard.

Some suitable test values are:

```
Input X coordinate: -2
Input Y coordinate: -1.25
Input Y range:      2.5
Threshold 1: 5
Threshold 2: 20
Threshold 3: 50
```

Once the display is complete, the host system's bell will be rung. Hit Enter, and the first prompt will reappear. You can then experiment with other sets of parameter values. A more interesting set of values is: −0.25, 0.8, 0.25, 10, 20, 50.

Use Ctrl-C when you want to stop the program.

Once you have the program working, you can make it run faster simply by plugging more T4 transputers into the network and reinvoking `fmandel.b4`.

7.6 Heterogeneous Networks

A flood-filled application compiled for the T4 and configured using the simple **master** and **worker** forms of task declaration may work on a mixed network of T4 and T8 processors if it uses only integer operations. This approach will not in general work for an application which uses floating-point operations, because of the incompatibilities between the T4 and T8 instruction sets.

Mixed networks of T4 and T8 processors are properly handled by an extension to the configuration file, like this:

```
task t4master file=mandelm4
task t8master file=mandelm8
task t4worker file=mandelw4 data=10k
task t8worker file=mandelw8 data=10k opt=stack
```

Separate tasks must be compiled and linked for T4 and T8 processors; the Parallel Fortran software ensures that the right task images are loaded into the right processors.

Again the names **t4master**, **t8master**, **t4worker** and **t8worker** are special, but the file names derived from them can be over-ridden by the FILE attribute, as above.

Note that it is possible to specify different memory optimisation options (e.g., **opt=stack** above) for the T4 and T8 variants of a task. This is useful because the T4 and T8 have different amounts of on-chip RAM.

If a **t4master** task is declared, a corresponding **t8master** task must also be declared, and similarly for the worker task.

Part III

Language Reference

Introduction

This Part describes the language recognised by the Parallel Fortran compiler. It is primarily intended for users with previous experience of Fortran programming.

The internationally accepted standard for Fortran (ANSI X3.9-1978 and ISO 1539-1980) (see [1]) is supported by the compiler. This standard is referred to variously throughout this publication as either the ANSI Standard or the Fortran 77 Standard. Parallel Fortran also supports various extensions to the Standard and these are identified in the text.

This Part includes chapters 8 to 16 of the *Parallel Fortran User Guide*. Chapter 8 is a general introductory chapter which describes the basic elements of the language, and chapters 9 and 10 describe the various types of data, their values, and how they are stored. Chapter 11 is concerned with expressions and chapter 12 with assignment statements. Chapters 13 and 14 describe the transfer of control within and between the units of a program respectively. Chapter 15 is concerned with format specifications, which are used in conjunction with the input and output facilities described in chapter 16.

The compiler's intrinsic functions, including those which are extensions, are described in appendix E.

Certain facilities have been included in the compiler in order to help those who are porting programs from earlier compilers. These features are described in appendix D.

Chapter 8

Fundamentals

This chapter provides general Fortran information. The chapter introduces basic terminology and outlines the structure of a Fortran program.

Parallel Fortran is based on the ANSI Fortran 77 standard as defined in *ANSI X3.9-1978*[1]. Extensions to the language have been provided as a transition aid from other Fortran dialects. These extensions are noted throughout this document and in the index.

Such extensions are allowable within the ANSI Fortran 77 standard since they do not conflict with the standard definition, but they should not be used in programs that are intended to be portable to other implementations of Fortran 77. The compiler issues a warning by default when any non-standard Fortran construct is used.

Fortran is a programming language designed primarily for the mathematical or scientific user. A Fortran program is written as a series of statements using symbolism analogous to that used in algebra. Many of these statements are readily intelligible to a programmer with mathematical training. For example, the Fortran expression

```
(A+B)/C
```

resembles a line of algebra and has a similar meaning. Each statement occupies at least one line of coding and can extend onto subsequent lines if necessary. Statements can be given identifying labels. Fortran provides facilities for the evaluation of common mathematical functions. The programmer need only write

```
Y = SIN(X)
```

and Fortran evaluates the sine function. Appendix E lists the standard procedures. You can write similar procedures for yourself as external functions.

A Fortran program normally executes in the order in which statements are written, but various control statements enable the programmer to specify that control branches to another statement with an identifying label, either unconditionally or if certain conditions are satisfied.

You can write Fortran programs as one or more program units and compile each program unit separately. One program unit is designated as the master program unit. This program unit controls the running of the program and passes control to other program units.

8.1 Character Set

The set of characters used in writing Fortran programs is:

- alphabetic:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
```

- numeric:

```
0123456789
```

- special characters:

```
+-*/=,.:()'$&\"_%
```

- the space (or blank) character. When necessary in this text, the symbol '␣' will be used to make explicit the location of a space.

No character other than these may be used except in character constants (see section 9.2), and in comment lines (see section 8.3.1.3).

Standard Fortran uses only upper case alphabetic characters. In Parallel Fortran, lower case is also accepted. Lower case alphabetic characters are equivalent to upper case characters except when they appear in character strings or Hollerith constants.

Alphabetic and numeric characters are referred to collectively as *alphanumeric* characters.

8.2 Program Structure

A program consists of *program units*. A program always has at least one program unit, called the *main program*, and may have one or more other program units that are called *subprograms*. Execution of the program starts in the main program and then control is passed between the main program and subprograms or between subprograms. For further details of transfer of control between program units see chapter 14.

There are three classes of subprogram:

1. Function subprograms
2. Subroutine subprograms
3. Block data subprograms

Function and *subroutine* subprograms provide a mechanism to assist the programmer in structuring programs in a meaningful way, and to allow common code to be conveniently accessed. These subprograms are described in detail in chapter 14.

Block data subprograms are used to give initial values to variables and arrays used in more than one program unit. They differ from other subprograms in that they can contain only certain non-executable statements (see section 8.3.4.2) and in that control is never passed to them. Block data subprograms are described in detail in section 10.3.2.

8.3 Program Unit Structure

8.3.1 Lines

A line in a program unit consists of 72 character positions. The character positions are numbered from 1 to 72. A statement occupies positions 7 to 72 of one or more lines. Any text following position 72 is ignored. If the compiler is invoked with the `/R` switch, the line length is extended to 132 character positions.

In Parallel Fortran a `TAB` character in the first position of a line can be used to skip past the statement label positions. If the character following the `TAB` character is a digit this is assumed to be in position 6, the continuation indicator position. Any other character following the `TAB` character is assumed to be in position 7, the start of a new statement. A `TAB` character in any other position of a line is treated as a space.

There are three classes of Fortran line.

8.3.1.1 Initial Line

An *initial line* has the following form:

- Positions 1 to 5 may contain a statement label (see section 8.3.3 below).
- Position 6 contains a space or the digit '0'.

- Positions 7 to 72 (or 7 to 132, if the /R switch is used) can contain the statement.

8.3.1.2 Continuation Lines

A *continuation line* has the following form:

- Positions 1 to 5 are blank.
- Position 6 contains any character other than '0' or a space. It is usual to number continuation lines consecutively from 1.
- Positions 7 to 72 (or 7 to 132, if the /R switch is used) contain the continuation of a statement.

In Parallel Fortran an alternative form is possible. In this case the first position of the line contains an ampersand '&', and the rest of the line forms the statement.

8.3.1.3 Comment Lines

Comment lines may be included in a program; such lines do not affect the program in any way but can be used by the programmer to include explanatory notes. The letter 'C' or an asterisk '*' in position 1 of a line designates that line as a comment line. The comment text is written in positions 2 onwards. A line containing only blank characters in positions 1 to 72 (or in all positions, if /R is specified) is also a comment line.

In Parallel Fortran an exclamation mark '!', either in position 1 or in any position from 7 onward, causes the rest of the line to be treated as a comment.

In Parallel Fortran, lines with a 'D' in position 1 are known as *debug comments*. Normally, such lines are treated as if the 'D' were a 'C', that is, as comments. However, if the compiler is invoked with the

switch **/D**, the **'D'** is treated as a space, so that the debug comment is compiled.

8.3.2 Statements

A statement consists of an initial line and, where necessary, up to 19 continuation lines.

Except as part of a logical **IF** statement, no statement may begin on a line that contains any part of the previous statement.

Blank characters may appear preceding, within or following a statement without changing the interpretation of the statement, except when they appear within character constants or the **'H'** or apostrophe **' '** format codes in **FORMAT** statements.

An **END** statement statement marks the end of a program unit. The statement consists of the three characters **'E' 'N' 'D'** in that order, in any of positions 7 to 72 of an initial line. All other positions from 1 to 72 must contain spaces. No other statement may have an initial line that appears to be an **END** statement.

8.3.3 Statement Labels

Any statement in a Fortran program may be identified by preceding it with a statement *label*.

A statement label is an unsigned integer in the range 1 to 99999. The numbers used as labels have no sequential significance. For example, the label 7 may occur after the label 9853. Labels may appear anywhere within columns 1 to 5. Blanks and leading zeros have no significance in labels.

All statement labels within any one program unit must be unique. Labels may be referred to only in the program unit in which they occur.

8.3.4 Categories of Statement

Each statement is classified as executable or non-executable.

Executable statements specify actions and form an execution sequence in a program.

Non-executable statements specify characteristics, arrangement, and initial values of data; contain format editing information; specify statement functions; classify program units; and specify entry points within subprograms. Non-executable statements are not part of the execution sequence. They may be labelled, but such statement labels must not be used to control the execution sequence.

8.3.4.1 Executable Statements

The following statements are classified as executable:

- Arithmetic, logical, statement label (`ASSIGN`), and character assignment statements
- Unconditional `GO TO`, assigned `GO TO`, and computed `GO TO` statements
- Arithmetic `IF` and logical `IF` statements
- Block `IF`, `ELSE IF`, `ELSE`, and `END IF` statements
- `CONTINUE` statement
- `STOP` and `PAUSE` statements
- `DO` statement
- `READ`, `WRITE`, `PRINT` and, in Parallel Fortran, `TYPE` and `ACCEPT` statements
- `REWIND`, `BACKSPACE`, `ENDFILE`, `OPEN`, `CLOSE` and `INQUIRE` statements

- In Parallel Fortran, **DECODE**, **ENCODE**, **DEFINE FILE**, and **FIND** statements
- **CALL** and **RETURN** statements
- **END** statement
- **END DO** statement

8.3.4.2 Non-executable Statements

The following statements are classified as non-executable:

- **PROGRAM**, **FUNCTION**, **SUBROUTINE**, **ENTRY** and **BLOCK DATA** statements
- **DIMENSION**, **COMMON**, **EQUIVALENCE**, **IMPLICIT**, **PARAMETER**, **EXTERNAL**, **INTRINSIC**, **SAVE** statements, and, in Parallel Fortran, **NAMELIST** and **VIRTUAL** statements
- **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **LOGICAL**, **CHARACTER** type-statements and, in Parallel Fortran, **DOUBLE COMPLEX** and **BYTE** type-statements
- **DATA** statement
- **FORMAT** statement
- Statement function statement

8.3.5 Order of Statements and Lines

Table 8.1 is a diagram of the required order of statements and comment lines for a program unit. Vertical lines delineate varieties of statements that may be interspersed. For example, **FORMAT** statements may be interspersed with statement function statements and executable statements. Horizontal lines delineate varieties of

<table>
<tr><td rowspan="5">Comment lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE or BLOCK DATA statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY</td><td rowspan="2">PARAMETER statements</td><td>IMPLICIT statements</td></tr>
<tr><td>Other specification statements</td></tr>
<tr><td rowspan="2">DATA statements</td><td>Statement function statements</td></tr>
<tr><td>Executable statements</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Table 8.1: Required Order of Statements and Comment Lines

statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements.

Within the specification statements of a program unit, **IMPLICIT** statements must precede all other specification statements except **PARAMETER** statements. Any specification statement that specifies the type of a symbolic constant must precede the **PARAMETER** statement that defines that particular symbolic constant; the **PARAMETER** statement must precede all other statements containing the symbolic constants that are defined in the **PARAMETER** statement.

ENTRY statements may appear anywhere except between a block **IF** statement and its corresponding **END IF** statement, or between a **DO** statement and the terminal statement of its DO-loop.

The last line of a program unit must be an **END** statement.

All statement function statements must precede all executable statements.

8.4 Names

In Fortran various items are identified by *names* chosen by the programmer. In standard Fortran 77 a name is a string of up to six alphanumeric characters of which the first must be alphabetic. Sometimes the first character has special significance (see section 9.3.1). Spaces normally have no significance in Fortran programs and so, for example, the following names are identical:

```
NAME1
N AME1
NAME 1
```

In Parallel Fortran symbolic names may include up to 31 alphanumeric characters, all of which are significant; and may include the characters '$' and '_', though not initially.

In general, a name has only one meaning within a program unit. The same name used in different program units does not in general refer to the same object except when it refers to a subprogram or common block name. There are three exceptions to these rules:

1. A common block name may also be a variable, array or statement function name.

2. A function subprogram name must also be a variable name within the function subprogram (see section 14.1).

3. The name of a variable used as the DO-variable of an implied-DO in a DATA statement may have any other meaning outside the implied-DO list.

Note: The term *symbolic name* is sometimes used instead of *name*.

Chapter 9

Data

This chapter is concerned only with the organisation of data in a Fortran program. The three permissible types of data are described, together with the possible methods of data specification. Data storage and input/output are described in chapters 10 and 16 respectively.

9.1 Data Values and Types

Values in Fortran can be classified as follows:

1. *Arithmetic* values. These can be further subdivided into:

    - *Integer* values, which are whole numbers. Such values are said to be of type integer and are held exactly in fixed point form in store.
    - *Real* values, which are numbers expressed as decimal fractions with exponents. Such values are said to be of type real and are held approximately in floating-point form in store.

- *Double precision* values, which are numbers held in the same form as real values but to a greater precision.
- *Complex* values, representing complex numbers. Such values are said to be of type complex and are held in store as a pair of real values, the first representing the real part and the second representing the imaginary part.
- In Parallel Fortran, *double complex* values, which are complex numbers each part of which is held as a double precision number.

2. *Logical* values, representing the values true or false. Such values are said to be of type logical.

3. *Character* values, representing strings of characters. Such values are said to be of type character, and their length is under the control of the programmer. In Parallel Fortran the length of character variables, array elements, or constants may be from 1 to 32767 characters.

9.2 Constants, Variables, and Arrays

Values can be made available for use in calculations in one of five ways:

1. As a *constant* value which can be written at the point in the program at which it is required (see section 9.2.1).

2. As a *symbolic constant* which has previously been named and defined with a value in a PARAMETER statement (see section 9.3.6).

3. In a *variable*. This is a named area of storage which can contain one item of data of a particular type, the type being determined by the variable name (see section 9.2.3) or by a type specification statement (see section 9.3).

4. In an *array*. This is a named area of storage which can contain a set of items of data of a particular type, the type being determined by the array name or by a type specification statement (see section 9.3). Each item of data within the set is called an *array element* (see section 9.2.4).

5. In a *character substring*. This is an unbroken portion of a variable or array element of type character (see section 9.2.5).

Variables, arrays, and array elements may be assigned initial values by use of **DATA** statements (see section 10.3.1) and may be assigned new values during the execution of the program.

9.2.1 Constants

There are six types of constant that can be used in standard Fortran: integer, real, double precision, complex, logical, and character. Integer, real, double precision, and complex constants are grouped together as arithmetic constants. Parallel Fortran adds another type of arithmetic constant, double complex.

9.2.1.1 Integer Constants

An *integer constant* is an optionally signed whole number written as a string of digits with no decimal points or exponents. Unsigned integer constants are assumed to be positive.

9.2.1.2 Real Constants

Real constants are numbers written containing a decimal point, an exponent or both. They may be signed or unsigned. If they are unsigned they are assumed to be positive. *Exponents* are written as the letter 'E' followed by a signed or unsigned integer. The integer represents a power of ten to which the constant is to be raised.

Thus real constants may take any of the following forms:

$\pm n.m$	$\pm n.m\mathtt{E}\pm a$
$\pm n.$	$\pm n.\mathtt{E}\pm a$
$\pm .m$	$\pm .m\mathtt{E}\pm a$
	$\pm n\mathtt{E}\pm a$

where n, m, and a are strings of digits, and $\pm$ is an optional sign, '+' or '-'.

9.2.1.3 Double Precision Constants

Double precision constants are numbers written containing an optional decimal point and an exponent. They may be signed or unsigned. If they are unsigned they are assumed to be positive. Exponents are written as the letter 'D' followed by a signed or unsigned integer. The integer represents the power of ten to which the constant is to be raised. Double precision constants take any of the following forms:

$$\pm n.m\mathtt{D}\pm a$$
$$\pm n.\mathtt{D}\pm a$$
$$\pm .m\mathtt{D}\pm a$$
$$\pm n\mathtt{D}\pm a$$

where n, m, and a are strings of digits, and $\pm$ is an optional sign, '+' or '-'.

9.2.1.4 Complex Constants

Complex constants are pairs of real or integer constants; the first constant corresponds to the real part of a complex number and the second corresponds to the imaginary part. Complex constants have the form (a,b), where a and b are constants and (a,b) represents the complex number $a+ib$. The form $-(a,b)$ is not a valid complex constant, and would have to be written $(-a,-b)$.

9.2.1.5 Double Complex Constants

A *double complex constant* is a complex number each part of which is held as a double precision number. It has the same form as a complex constant except that *a* and *b* are double precision constants.

Double complex constants are not available in standard Fortran.

9.2.1.6 Logical Constants

There are two *logical constants*, representing the values true and false. They have these forms:

```
.TRUE.
.FALSE.
```

9.2.1.7 Hollerith Constants

In Parallel Fortran *Hollerith constants* may be used for data initialisation in **DATA** statements and in the argument list of a **CALL** statement. In **DATA** statements, non-character variables and array elements may be initialised by Hollerith constants and each constant must have a length which is less than or equal to the length of the item. If the constant is shorter than the item, it is extended on the right with blanks.

Variables and array elements which are not of type character may alternatively be assigned Hollerith data by using the **Aw** edit descriptor in a formatted **READ** statement (see chapter 15). This facility is an extension to the ANSI Standard. The **Aw** edit descriptor may also be used to output variables and array elements which contain Hollerith data. Non-character arrays are also permitted in Parallel Fortran to define a format specification; see section 15.2.2 for further information and section 15.4 for examples.

An actual argument in a subroutine reference may be a Hollerith constant. The corresponding dummy argument must be of type integer, real, double precision, or logical. If the length of the constant is one to four bytes then a four byte argument is passed (blank characters being added to the right if necessary). If the length of the constant is five to eight bytes then an eight byte constant is passed.

Hollerith constants are not allowed in the ANSI Standard; they are not compatible with variables or array elements of type character and they may not be used to initialise, or assign new values to, such variables.

9.2.1.8 Hexadecimal Constants

In Parallel Fortran *hexadecimal constants* may be used to initialise logical, integer, byte or real variables. Two forms of the constant are supported:

X'*value*'
'*value*'**X**

where *value* is a sequence of hexadecimal digits (**0**–**9**, **A**–**F**).

Hexadecimal constants may only appear in **DATA** statements, or in the special form of type statement which allows data initialisation (see section 10.3.1.8). They may not be used in executable statements.

9.2.1.9 Octal Constants

Octal constants may be used to initialise logical, integer, byte or real variables. Use of these constants is restricted to **DATA** statements and the special form of type statement which allows data initialisation (see section 10.3.1.8). The form of an octal constant is:

O'*value*'

where *value* is a string of octal digits (0–7).

Octal constants are not allowed in the ANSI standard.

9.2.1.10 Binary Constants

Binary constants may be used to initialise logical, integer, byte or real variables. The form of a binary constant is:

B'*value*'

where *value* is a string of binary digits (0, 1). Binary constants are not allowed in the ANSI Standard, and they may not appear in an executable statement. Their use is restricted to **DATA** statements and the special form of type statement which allows data initialisation (see section 10.3.1.8).

9.2.1.11 Character Constants

A *character constant* is a non-empty string of any characters, delimited by being enclosed in apostrophes ' '. In Parallel Fortran a character constant may alternatively be enclosed in double quotes " ".

If a string enclosed in apostrophes itself contains an apostrophe, this must be represented by two apostrophes to distinguish it from a delimiting apostrophe. In Parallel Fortran the same applies when a string is delimited by double quotes; if another double quote appears in the string it must be repeated. But if a string is delimited by one sort of marker, then the other can appear in the string without needing to be repeated. The backslash escape character '\' described below provides another mechanism to allow embedding quotes in strings.

The length of a character constant is the number of characters which appear between the delimiting apostrophes or quotes, except that

each pair of consecutive apostrophes or quotes counts as a single character.

The following are examples of valid character constants:

```
'C(1)='
'MUSTARD AND CRESS'
'ISN''T'
```

Using the alternative double quote in Parallel Fortran, the following would be valid character constants:

```
"RADIUS ="
"ISN'T"
```

For compatability with C usage, the backslash '\' is allowed in Parallel Fortran as an escape character. It denotes that the following character in the string has a significance which is not normally associated with the character. The effect is to ignore the backslash character, and either substitute an alternative value for the following character or to interpret the character as a quoted value. The escape characters recognised, and their effects are as follows:

Escape Character	Effect
\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	double quote (does not terminate a string)
\\	backslash
\x	x, where x is any other character

For example,

```
'ISN\'T'
```

is a valid string.

The backslash is not counted in the length of the string.

9.2.2 Symbolic Constants

A *symbolic constant* is a constant value that is identified by a name (see section 8.4). The value associated with the symbolic constant is defined in a **PARAMETER** statement (see section 9.3.6) which must appear before any use is made of the name to represent a value. The type of a symbolic constant is determined in the same way as for a variable (see section 9.2.3 and section 9.3).

9.2.3 Variables

A *variable* is an item of data that is identified by a name (see section 8.4). Values can be assigned to variables during the execution of a program. The value assigned to a variable at any time is made available to the program when a reference is made to the variable name.

In general, a particular variable will be available in only one program unit. A name used for a variable in one program unit may be used for an entirely different variable in another program unit.

There are six types of variables in standard Fortran 77: integer, real, double precision, complex, logical, and character. Parallel Fortran adds two more: byte and double complex. The ranges of values these types can take are the same as for the corresponding types of constants (see section 9.2.1) with the following exceptions.

- Real, double precision and complex arithmetic may result in special values as defined by the IEEE standard for floating point numbers[9] (see section 11.1.10).
- Byte variables have integer values, but these values must be in the range -128 to $+127$.

If the name chosen for a variable begins with one of the letters 'I' to 'N' inclusive, then the variable will be assumed to be of type integer. Otherwise it will be assumed to be of type real. However, the programmer can override this convention by specifying, in a type specification statement, the type the variable is to be (see section 9.3).

For example, variables with names such as **INT**, **LIST**, **NUMBER** or **J322** would be assumed to be of type integer unless otherwise specified. Variables with names such as **AREA**, **SUM** or **R147** would be assumed to be of type real unless otherwise specified.

This method of defining the types of variables can result in small coding errors creating unwanted variables, which can be hard to track down. For this reason, the Parallel Fortran compiler can be invoked with the **/U** switch. This stops variables from being defined in this automatic way, and obliges the programmer to define all variables explicitly. See section 9.3 below for a further discussion of this.

9.2.4 Arrays

Sets of data items of the same type can be processed as *arrays*. A single name, the array name, is chosen to identify the set, and individual items are called the array *elements* (see section 8.4 for further details concerning names). Arrays may have one or more dimensions. For example, the matrix **A**:

```
A(1,1) A(1,2) A(1,3) A(1,4)
A(2,1) A(2,2) A(2,3) A(2,4)
```

could be treated as a two-dimensional array with eight elements. Arrays may have up to seven dimensions.

There are six types of arrays in standard Fortran 77: integer, real, double precision, complex, logical, and character. Parallel Fortran adds two more: byte and double complex. The type of an array is determined in the same way as the type of a variable, and each element of the array has this same type.

In some contexts an array may be referred to as a whole by specifying the array name. In other contexts individual elements may be referred to by an array element reference which takes the form of the array name followed by a subscript list enclosed in parentheses. A *subscript list* is an ordered set of subscript expressions separated by commas, one subscript expression for each dimension of the array. A *subscript expression* may be an arithmetic expression (see chapter 11) which in standard Fortran must be of type integer. In Parallel Fortran, however, subscript expressions may also be of type real.

The compiler allocates storage to the array as instructed by an *array declarator* (see section 10.2.2). The array declarator and the subscript expressions given in the array element reference are used to calculate the position in store that is occupied by the specified element. The order in which array elements are held in store is specified in section 10.1.2.

Each subscript expression, when evaluated, must have a value within the declared bounds for that subscript.

The following are examples of valid array element references, with explanations:

`TABLE(7)` Element (7) of the one dimensional array **TABLE**

`MAT(I,I+1)` If I is an integer variable with the value 7, this reference is to element (7,8) of the two-dimensional array **MAT**.

`VECTOR(IFUN(J,3))`
If **IFUN** is an integer external function or statement function requiring two actual arguments and **VECTOR** is a one dimensional array, then the function is evaluated to give the array element required.

9.2.5 Character Substrings

A *character substring* is an unbroken portion of a character scalar or array element and is a variable of type character. It may be assigned values and referenced, and is identified by a substring name in one of these forms:

$c(p_1:p_2)$
$a(k_1,k_2,\ldots)(p_1:p_2)$

where:

c is a character variable name.

$a(k_1,k_2,\ldots)$ is a character array element name.

p_1 and p_2 are integer expressions and are known as *substring expressions*.

The value p_1 specifies the leftmost character position of the substring, and p_2 specifies the rightmost character position. The values of p_1 and p_2 must be such that

$$1 \leq p_1 \leq p_2 \leq s$$

where s is the length of the character variable c or the array element $a(k_1,k_2,\ldots)$. If p_1 is omitted then the value of 1 is assumed, and if p_2 is omitted then the value of s is assumed; both p_1 and p_2 may be omitted.

9.3 Type Specification

In Fortran all constants, symbolic constants, variables, arrays and functions must be identified as being of particular *types* so that they can be stored and processed correctly. The type of a constant is indicated by the way the constant is written. The type of a symbolic

constant, variable, array, or function may be defined in any of three ways:

1. Predefined specification
2. IMPLICIT specification
3. Explicit specification statements

Explicit statements override IMPLICIT specifications, which in turn override predefined specifications.

9.3.1 Predefined Specification

Any symbolic constants, variables, arrays or functions whose names are not mentioned in a type specification statement and whose initial letter is not mentioned in an IMPLICIT statement (see section 9.3.2) will be assumed to be of type integer or real according to the following rules:

- If the name of the symbolic constant, variable, array or function begins with one of the letters I, J, K, L, M or N the compiler assumes the symbolic constant, variable, array, or function to be of type integer.
- If the name begins with any other letter the quantity is assumed to be of type real.

Some examples of predefined type variable names are given in section 9.2.3.

Parallel Fortran has a compile-time switch /U which stops the compiler from predefining the types of symbolic constants, variables, arrays or functions in this way. When a program is compiled with this switch, everything must be defined with the IMPLICIT statement or one of the explicit type specification statements, as described next.

9.3.2 The IMPLICIT Statement

The **IMPLICIT** statement provides a means of overriding the Fortran convention of predefined specification for the types of symbolic constants, variables, arrays and functions. This takes effect for the whole of the current program unit unless overridden by explicit type statements. The statement takes the form:

IMPLICIT $type_1(a_1,a_2,\ldots),\ldots,type_n(a_m,a_n,\ldots)$

where:

type$_i$ is one of: **INTEGER**, **BYTE**, **REAL**, **DOUBLE PRECISION**, **LOGICAL**, **COMPLEX**, **DOUBLE COMPLEX**, or **CHARACTER***s.

$a_1,a_2,\ldots$ and $a_m,a_n,\ldots$
are lists of single alphabetic characters separated by commas, or a range of alphabetic characters in sequence, separated by a minus sign. The same letter may not appear singly, or within a range of characters, more than once in a subprogram.

s is the length of the character entities and is either an unsigned, non-zero integer constant, or an integer constant expression enclosed in parentheses and with a positive value. s (together with the preceding *) is optional and, if omitted, the length is one.

After this statement has been processed, all symbolic constants, and variable, array or function names beginning with the characters $a_1,a_2,\ldots$ are implicitly of type *type*$_1$ and all symbolic constants, and variable, array or function names beginning with $a_m,a_n,\ldots$ are implicitly of type *type*$_n$ unless the specification is overridden by an explicit specification statement.

A program unit may contain more than one **IMPLICIT** statement, but **IMPLICIT** statements must precede all other specification statements except **PARAMETER** statements. For a subprogram, **IMPLICIT** statements can specify the type of the parameters to the subprogram, and

of the function name for a function subprogram, unless their types are specified in an explicit type specification statement.

Here are two examples of the IMPLICIT statement.

```
IMPLICIT REAL(A-D,X,Z),LOGICAL(L)
```

This statement specifies that all variables whose names begin with A, B, C, D, X or Z that do not appear in explicit type statements are to be real. Similarly all variables whose names begin with L are assumed to be logical.

```
COMPLEX FUNCTION BACH(THEME,FUGUE)
IMPLICIT DOUBLE PRECISION(A-H)
```

The overall effect of these two statements is that the parameter FUGUE will be of type double precision and the function BACH will be of type complex. The parameter THEME is assumed to be type real by virtue of its initial letter T.

9.3.3 The IMPLICIT NONE Statement

This statement, provided in Parallel Fortran, overrides all the predefined type specification provisions of Fortran. If an IMPLICIT NONE statement is included in a program unit then all the names in that unit must have their type explicitly declared. A program unit that includes an IMPLICIT NONE statement may not include any other IMPLICIT statements.

9.3.4 The IMPLICIT UNDEFINED Statement

This statement, provided in Parallel Fortran, has similar effects to IMPLICIT NONE. It has the form:

```
IMPLICIT UNDEFINED (a1-a2)
```

where a_1 and a_2 are alphabetic characters. This statement overrides the predefined typing mechanism for names beginning with the let-

ters a_1 to a_2. For example, variables with names beginning with the letters I to N would normally, unless explicitly specified, be of type integer. But the statement

```
IMPLICIT UNDEFINED(L-N)
```

overrides the automatic classification as integer for variables beginning with the letters 'L', 'M', and 'N'. If any variable names begin with these letters, their types would have to be explicitly specified.

9.3.5 Explicit Type Specification Statements

Explicit type specification statements are used to confirm or override the predefined or implicit type specification, and optionally to give dimension information for arrays.

The appearance of the name of a symbolic constant, variable, array, external function or statement function specifies the data type for that name for all appearances in the program unit. Within a program unit a name must not have its type explicitly specified more than once. A type statement which confirms the type of an intrinsic function (listed in appendix E) is permitted, but is not necessary. The appearance of a generic function name (listed in appendix E) (see section 14.1.2.1) in a type statement does not necessarily remove the generic properties of that function. Explicit type specification statements may also, in Parallel Fortran, assign initial values to data items. This initialisation is defined in the same manner as for a DATA statement (see sections 10.3.1 and 10.3.1.8).

9.3.5.1 Arithmetic and Logical Type Statements

These statements take the form:

type $var_1(dim_1), var_2(dim_2), \ldots, var_n(dim_n)$

where:

each $type_i$ is one of: **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **LOGICAL**, or, in Parallel Fortran, **DOUBLE COMPLEX** or **BYTE**

each var_i is a symbolic constant, variable, array, function or dummy procedure name (see section 14.1).

each (dim_i) is optional and gives dimension information for arrays (see section 10.2.2).

Here are some examples of explicit type specifications statements.

```
REAL A,B(10),C,D
```

This statement declares **A**, **C** and **D** to be real, and **B** to be a real array with 10 elements.

```
INTEGER FRED,JIM,UNCLES(5)
```

This statement declares the variables **FRED** and **JIM**, of type integer. In addition, the integer array **UNCLES** is declared, which has five elements.

```
DOUBLE PRECISION HEXNO,INTNO
```

This statement declares two double precision variables, **HEXNO** and **INTNO**.

```
LOGICAL L,BOOLE
```

This statement declares two logical variables, **L** and **BOOLE**.

In Parallel Fortran the following data type specifications are also allowed:

```
LOGICAL*4
INTEGER*4
REAL*4 REAL*8
COMPLEX*8 COMPLEX*16
```

In each case, the number following the '*' indicates the number of bytes allocated.

9.3.5.2 The CHARACTER Type Statement

This statement is written:

CHARACTER$*s$ $var_1(dim_1)*s_1, var_2(dim_2)*s_2, \ldots, var_n(dim_n)*s_n$

where:

each var_i is a symbolic constant, variable, array, function or dummy procedure name (see section 14.1).

each (dim_i) is optional and gives dimension information for arrays (see section 10.1.2).

$*s$ and each $*s_i$
are optional *length specifications* (numbers of characters) of a character variable, character array element, character symbolic constant, or character function. Each s is one of the following:

- An integer constant, in the range 1 to 32767;
- An integer constant expression enclosed in parentheses and with a positive value;
- An asterisk in parentheses.

A $*s$ immediately following the word **CHARACTER** is the length specification for each entity in the specification not having one of its own. A length specification immediately following an entity applies only to that entity: for an array the length specification is for each element of that array. If a length is not specified for an entity, its length is one. If a length is specified for an entity declared in the statement, the length specification must be a positive non-zero integer constant expression, unless the entity is an external function, a dummy argument of an external subprogram or a character symbolic constant.

If a dummy argument (see section 14.1) has a length '(*)' declared, the dummy argument assumes the length of the associated actual argument for each reference of the subprogram. If the associated

actual argument is an array name, the length assumed by the dummy argument is the length of an array element in the associated actual argument array.

If an external function has a length '(*)' declared in a function subprogram, the function name must appear as the name of a function in a **FUNCTION** or **ENTRY** statement in the same subprogram. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit.

The length specified for a character function in the program unit that references the function must be an integer constant expression and must agree with the length specified in the subprogram that specifies the function. There is always agreement of length if '(*)' is specified in the subprogram that specifies the function.

If a character symbolic constant has a length '(*)' declared, the symbolic constant assumes the length of its corresponding constant expression in a **PARAMETER** statement.

The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression.

Example:

```
CHARACTER CHAR,BUFF*80
```

This statement declares two character variables **BUFF** and **CHAR**. **CHAR** occupies one character (the default length) and **BUFF** occupies 80 characters.

9.3.6 The PARAMETER Statement

A **PARAMETER** statement is used to define the value of a *symbolic constant*. The statement has the form:

PARAMETER ($name_1$=$expr_1$, $name_2$=$expr_2$, ... , $name_n$=$expr_n$)

where:

each $name_i$ is a symbolic constant name.

each $expr_i$ is a constant expression.

If $name_i$ is of type integer, real, double precision, complex, or double complex, the corresponding $expr_i$ must be an arithmetic constant expression. If $name_i$ is of type character or logical, the corresponding $expr_i$ must be a character constant expression or a logical constant expression respectively.

Each $name_i$ is the name of a symbolic constant that is defined by the value of its corresponding $expr_i$ in accordance with the rules for assignment statements (see chapter 12). No *name* may be defined more than once in any program unit.

If any *name* is not to have the type specified implicitly then its type must be specified by a type-statement (see section 9.3.5) or an **IMPLICIT** statement (see section 9.3.2) before its appearance in a **PARAMETER** statement. If the length specified for a symbolic constant of type character is not the default length of one, then its length must first be given in an **IMPLICIT** or type statement. Its length cannot be changed subsequently.

Once a symbolic constant has been defined it may be used in any subsequent statement in the same program unit as an element of an expression or in a **DATA** statement, but not as part of a format specification or as part of another constant.

Chapter 10

Storage of Data

This chapter deals with the storage of data. It describes how quantities are held in store according to their type and then describes the various non-executable statements concerned with allocating storage and assigning initial values to variables.

Specification of type is described in chapter 9 and the order in which the non-executable statements described in this chapter must occur is given in section 8.3.5.

10.1 Storage Requirements

The standard unit of storage is a *byte*, which consists of 8 binary digits. The amounts of storage required under Parallel Fortran by the various types of data are defined below.

10.1.1 Constants and Variables

10.1.1.1 Integer

An integer constant or variable occupies four bytes. An integer value is held in twos complement form, and may range from -2^{31} to $+2^{31}-1$, that is, from −2147483648 to +2147483647.

Some examples of valid integer constants are:

```
0               5678          2147483647
-2147483648     -255          -0
+0              +5678         +2147483647
```

10.1.1.2 Byte

A byte variable occupies 1 byte. The value held in a byte variable must be an integer in the range −128 to +127. Byte variables are an extension to the ANSI Standard.

10.1.1.3 Real

A real constant or variable occupies four bytes. A real value is held as a normalised floating point number in accordance with the IEEE floating point format (see *IEEE Standard for Binary Floating-Point Arithmetic*[9]) and may range from $+2^{-126}$ to approximately $+2^{+128}$, that is, approximately, from $+1.1754945 \times 10^{-38}$ to $+3.402823 \times 10^{+38}$. Some examples of valid real constants are:

```
1.23            -1.23           +1.23
0.              .0001234        667744.

1.23E2          +1.23E2         -1.23E2
123456.E5       1.23E30         1.23E+33
1.23E+3         0.E0            1.23E0
1.23E+03

-1.23E30        -1.23E10        +123E10
+123E10         123456789E-34   0E0
```

Also see section 9.2.1.2.

10.1.1.4 Double Precision

A double precision constant or variable occupies eight bytes. A double precision value is held as a normalised floating point number in accordance with the IEEE floating point format and may range from $+2^{-1022}$ to $+2^{1024}$, that is, approximately from $+2.2250738 \times 10^{-308}$ to $+1.7976931 \times 10^{308}$.

10.1.1.5 Complex

A complex number consists of a real part and an imaginary part. (The word real, in the term real part, is not used in the sense of section 9.2.1.2.)

A complex constant or variable occupies eight bytes. It consists of either a pair of real (in the sense of section 9.2.1.2) constants, or a pair of integer constants; the first of the pair is the real part and occupies the first four bytes and the second is the imaginary part which occupies the second four bytes. Some examples of valid complex constants are:

```
(3.75,-2.100)
(0.,0.)
(-2.75E+2,7.1E-2)
```

10.1.1.6 Double Complex

A double complex constant or variable occupies sixteen bytes. It has the same form as a complex constant except that the real and imaginary parts are double precision constants, as described in section 9.2.1.3. This form of constant is an extension to the ANSI Standard permitted by Parallel Fortran.

10.1.1.7 Logical

A logical constant or variable occupies four bytes.

10.1.1.8 Character

Each character in the character constant or variable occupies one byte.

A character constant or variable may comprise from 1 to 32767 characters.

10.1.2 Arrays

Arrays can take the same types as variables. Each of the array elements has the same type as the array and is allocated the standard amount of storage for a variable of that type. The type of the array depends on its name (see section 9.3) unless otherwise specified.

The number of dimensions of each array and their sizes must be specified once, and only once, in a `DIMENSION` statement, `COMMON` statement (see sections 10.2.2 and 10.2.3 or an explicit type statement (see section 9.3.5). Individual array elements may in general be referred to by giving the array name and a list of subscript expressions, one subscript expression for each dimension (see section 9.2.4).

However, it is sometimes necessary for the programmer to know how the elements of an array are arranged in store. They are stored consecutively so that the left-hand subscript varies most rapidly and successive subscripts vary less rapidly. For example, the following two **DIMENSION** statements:

```
DIMENSION A(3,2)
DIMENSION B(2,3,4)
```

would result in the two arrays being stored as follows:

```
A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)

B(1,1,1) B(2,1,1) B(1,2,1) B(2,2,1) B(1,3,1) B(2,3,1) B(1,1,2)
B(2,1,2)
B(1,2,2) B(2,2,2) B(1,3,2) B(2,3,2) B(1,1,3) B(2,1,3) B(1,2,3)
B(2,2,3)
B(1,3,3) B(2,3,3) B(1,1,4) B(2,1,4) B(1,2,4) B(2,2,4) B(1,3,4)
B(2,3,4)
```

10.1.3 Character Storage

Characters may be held in a variable or array element of type character. Character values may be given to variables or array elements in four ways:

1. By assigning a character value in a **DATA** statement (see section 10.3.1);

2. By specifying a character constant as an actual argument of a subroutine call or of a function reference;

3. By using an **A** format description in conjunction with a **READ** statement (see section 15.3.1.7);

4. By specifying a character constant or expression on the right-hand side of a character assignment statement.

10.2 Allocation of Storage

10.2.1 General Considerations

This section discusses various statements used for storage allocation.

COMMON statements (see section 10.2.3) allow the same area of storage to be accessed by a number of different program units. This allows values assigned in one program unit to be used in other units.

EQUIVALENCE statements (see section 10.2.4) allow the same storage space to be used for more than one variable or array within one program unit.

DIMENSION statements (see section 10.2.2) are used for declaring arrays (see also section 10.1.2).

10.2.1.1 Variables

It is not necessary to specify every variable name in a non-executable statement in order that storage will be reserved for it. Variable names may be mentioned in various non-executable statements but this will always be for some purpose other than merely informing the compiler of the existence of the variable. If a variable name is not mentioned in a non-executable statement, when the name is first encountered in an executable statement, the compiler will assume the variable to be of the type given by its initial letter (see section 9.3.1) and will implicitly allocate the amount of storage accordingly.

Note that the Parallel Fortran compiler can be invoked with a **/U** switch if it is desired to stop storage being allocated automatically in this way. If this is done, every variable name must have its type defined with an **IMPLICIT** statement or an explicit type specification statement. This can be useful in tracking down programming errors.

10.2.1.2 Arrays

Arrays must be specifically defined so that the compiler is informed of the number of dimensions and the size of each dimension. The definition is given by means of an array declarator, whose form is specified in section 10.2.2. Array declarators may be given in type specification statements, **COMMON** statements and **DIMENSION** statements, but a particular array declarator may only be given once in each program unit. An array name on its own does not constitute an array declarator and thus, for example, the array name on its own could occur in a **COMMON** statement and a type specification statement if the declarator were given in a **DIMENSION** statement.

10.2.2 The DIMENSION Statement

The **DIMENSION** statement is used to declare names as being array names and to specify the number of dimensions of the array and the size of each dimension, so that the appropriate amount of storage can be allocated to the array. The statement has the form

DIMENSION $aname_1(dim_1), aname_2(dim_2), \ldots, aname_n(dim_n)$

where:

each $aname_i$ is an array name.

each dim_i defines the number of dimensions and the number of elements in each dimension of the corresponding array. Each dim_i takes the form

$$l_1:u_1,\ l_2:u_2,\ \ldots,\ l_z:u_z$$

where:

each l_i and u_i are the lower and upper bounds of dimension i.

z specifies the number of dimensions, which should be in the range 1 to 7.

The dimension bounds are arithmetic expressions in which all constants, symbolic constants, and variables are of type integer. Integer variables may appear in dimension bound expressions only for adjustable array specifications. A dimension bound expression must not contain a function or array element reference. The upper dimension bound of the last dimension in an assumed size specification may be an asterisk (see section 14.3.3.2).

The value of either bound may be positive, negative or zero provided that the upper bound is not less than the lower bound. If the lower bound is not specified it is assumed to have the value one. An upper bound specified as an asterisk always has a value greater than or equal to the lower bound.

Each $aname_i(dim_i)$ is an *array declarator.* A declarator for each array used in a program unit must be given once and only once in that program unit, in either a **DIMENSION** statement, a type specification statement or a **COMMON** statement. A declarator appearing in a **COMMON** statement may not contain dimension sizes specified by integer variable names or an asterisk.

For example, the statement

```
DIMENSION A(10),B(1:500,0:9)
```

declares **A** as a one-dimensional array with ten elements and **B** as a two-dimensional array, one dimension having 500 elements and the other dimension having ten elements.

For compatibility with some other Fortran compilers, Parallel Fortran also provides the **VIRTUAL** statement. This has exactly the same syntax and effect as the **DIMENSION** statement.

10.2.3 The COMMON Statement

The **COMMON** statement enables areas of storage, known as *common blocks*, to be used in more than one program unit of a program. Thus, values obtained in one program unit can be used in other program units (see section 10.2.3.4).

The statement has the form:

COMMON /*cbname*$_1$/$a(d_1),b(d_2),\ldots,$/*cbname*$_i$/$c(d_3),d(d_4),\ldots$

where:

each /*cbname*$_i$/
: is an optional parameter specifying a common block name (see section 10.2.3.1 below).

$a,b,\ldots$
: are variable or array names. In a subroutine or function these must not be dummy arguments (see section 14.1). If any variable or array is of type character then all variables and arrays in that common block are required by the ANSI Standard to be also of type character. This restriction is relaxed by Parallel Fortran.

each (d_i)
: is an optional parameter giving dimension information for arrays.

An optional comma may appear after a list of variable or array names and before the slash '/' prefacing another common block name.

10.2.3.1 Common Block Names

Any number of common blocks may be used in a program. If desired, one common block may be used without a name: such a block is referred to as the *blank* common block. All other blocks must be given a block name chosen to obey the rules for names, given in

section 8.4. No intrinsic function name (see appendix E) may be used as a common block name nor may the name of a constant or of any program unit of the program be used. There are no other restrictions on common block names (and thus duplication of variable and other names by common block names is permissible).

If a blank common block is used, two consecutive slashes '**//**' must precede the names of variables and arrays in the block unless the blank common block is the first block given in the **COMMON** statement, in which case the two slashes may be omitted.

The same block name may appear more than once in a **COMMON** statement or in a subprogram; see section 10.2.3.3 below.

Here are some examples of the use of the **COMMON** statement.

```
COMMON /BLOCK 1/VALUE,ARRAY/BLOCK 2/X,Y,Z
```

The variable **VALUE** and the array **ARRAY** are held in a common block named **BLOCK1**, and variables **X**, **Y**, and **Z** are held in a common block named **BLOCK2**. The dimensions of the array **ARRAY** must be declared in a type statement or a **DIMENSION** statement, since they have not been given in the **COMMON** statement.

```
COMMON COUNT,TABLE,RESULT/AREA/SUB,ITEM
COMMON /AREA/SUB,ITEM//COUNT,TABLE,RESULT
```

These two statements have the same effect; **COUNT**, **TABLE**, **RESULT** are held in the blank common block and **SUB** and **ITEM** are held in common block **AREA**.

```
COMMON A,B(20)/COMM/D(40),E,I//X,Y,Z
```

This statement declares the variables **A**, **B**, **X**, **Y** and **Z** to be in the blank common block, and **D**,**E**,**I** to be in the block **COMM**. If later, the statement

```
COMMON /COMM/A1,A2,A3(17)/ON/BA,BC
```

occurs, then the variables **A1**, **A2**, **A3** are added to the end of block **COMM** and a new block **ON** is created with variables **BA**, **BC** (see section 10.2.3.3).

10.2.3.2 Storage of Variables

The size of a common block is the sum of the storage required for each element within it. However, program units which are to be executed together must specify the same sizes for named common blocks which they share. Variables are allocated consecutive units of storage in the order in which they occur in the program.

10.2.3.3 Multiple References within a Program Unit

It is permissible for a common block name (or a reference to the blank common block) to occur more than once in a COMMON statement or in more than one COMMON statement in the same program unit. For example, the statement

```
COMMON X,Y,CHECK/RESULT/A(10)//SUM,AREA/RESULT/B(10),C
```

is equivalent to the following two statements occurring in the same program unit:

```
COMMON X,Y,CHECK/RESULT/A(10)
COMMON SUM,AREA/RESULT/B(10),C
```

Either of these examples will have the same effect as the statement

```
COMMON X,Y,CHECK,SUM,AREA/RESULT/A(10),B(10),C
```

That is, variable and array names are allocated to the common block indicated in the order in which they occur in the program unit.

10.2.3.4 Using the COMMON Statement

The common block is an area of storage that will be used in more than one program unit of the program. The name of the block must appear in a COMMON statement in each program unit in which the block is to be used. The names of the variables and arrays within

the block may be the same in different program units, but need not be. Thus if the two statements

```
COMMON X,Y,CHECK/RESULT/A(10)
COMMON XCOORD,YCOORD,CH/RESULT/ARRAY(10)
```

occurred in two different program units, the variables **X**, **Y**, **CHECK** used in the first program unit would occupy the same storage area as **XCOORD**, **YCOORD** and **CH** used in the second program unit. The storage area is called the blank common block. Similarly **A(10)** in the first program unit and **ARRAY(10)** in the second program unit will share the storage area of common block **RESULTS**. Usually, the variable and array names included in a given common block in one program unit will correspond in number and type to those used in all other program units in which the block is referenced.

For the initialisation of named common blocks, see section 10.3. For the equivalencing of common items, see section 10.2.4.3.

10.2.4 The EQUIVALENCE Statement

The **EQUIVALENCE** statement enables variables and array elements of the same or different types used in one program unit to share storage space. Thus, the amount of storage used by a program unit can be reduced. There is no mathematical equivalence, simply a sharing or saving of space. The statement has the form

EQUIVALENCE $(a_1, a_2, a_3, \ldots), \ldots, (n_1, n_2, n_3, \ldots)$

where $a_1, a_2, a_3, \ldots, n_1, \ldots$ are variables, array elements, arrays or character substrings. They must not be function names or dummy arguments of functions or subroutines.

This statement causes all the elements of each list of variables, a_1 to a_m and n_1 to n_m to share the same storage area; that is, $a_1, a_2, \ldots, a_m$ will share one area and $n_1, n_2, \ldots, n_m$ will share another.

The subscripts of an array element used in this statement must be integer constant expressions and must correspond in number to the dimensions of the array.

In Parallel Fortran you may express an element of a multidimensional array in an **EQUIVALENCE** statement as a singly-subscripted reference, provided that the missing subscripts have the default value one. The compiler will assume a value of one for the missing subscripts.

10.2.4.1 Arrays in EQUIVALENCE Statement

Since array elements are stored in a fixed order, an equivalence between two elements of different arrays effectively defines an equivalence between other elements. It is important not to contradict these equivalences by further **EQUIVALENCE** statements.

10.2.4.2 Equivalencing Items of Different Types or Length

Variables, arrays and array elements except of type character may be equivalenced with other such items of different types, but wherever one of the equivalenced items of type *type1* is assigned a value of type *type1* then the value of another item of type *type2* that is equivalenced to the first item becomes undefined and should not be referred to in an expression until it has been assigned a value of type *type2*.

If equivalenced items occupy different amounts of store (for example if a real variable and a complex variable are equivalenced), the starts of the items are aligned. For example, the statement

```
EQUIVALENCE(A,B)
```

where **A** is of type complex and **B** is of type real will cause **B** to share the first four bytes of the storage allocated to **A**.

In standard Fortran an entity of type character may only be equivalenced to another entity of type character. The lengths of the

equivalenced entities do not need to be the same. In Parallel Fortran entities of type character may be equivalenced to entities of any other type.

10.2.4.3 Equivalencing Common Block Items

If one of the items given in the **EQUIVALENCE** statement appears in a common block then none of the other items given in the same list in the **EQUIVALENCE** statement may appear in the same or any other common block in the program unit.

Since an array element may be equivalenced to a variable in common, the implicit equivalencing of the rest of the array may extend the size of the common area. Such lengthening of a common block can take place only beyond the last entry in the block and not before the first entry. For example, the following series of statements:

```
COMMON /BLOCK/A(10),B,C,I,J
DIMENSION TABLE(3,4),ARRAY(4,4)
EQUIVALENCE(A(1),TABLE(1,1))
```

would result in the 12 elements of **TABLE** sharing the common storage area with the 10 elements of array **A** and the two variables **B** and **C**. If the **EQUIVALENCE** statement were replaced by

```
EQUIVALENCE(A(1),ARRAY(1,1))
```

then **ARRAY** would share storage with **A**, **B**, **C**, **I** and **J**, and the common block would be extended to hold the remaining elements of **ARRAY**. The **EQUIVALENCE** statement could not validly be replaced by

```
EQUIVALENCE(ARRAY(2,2),A(5))
```

since if this were implemented it would result in the common block **BLOCK** being extended before the first item in the block. Element **ARRAY(2,2)** is the sixth item of array **ARRAY**; if it were to be equivalenced with **A(5)** then **ARRAY(2,1)** would by implication be equivalenced with **A(1)** and the block would have to be extended before

`A(1)` to give storage to `ARRAY(1,1)`. This therefore is an invalid `EQUIVALENCE` statement.

10.3 Assignment of Initial Values

Initial values may be assigned to variables, arrays, array elements and substrings by using `DATA` statements. In Parallel Fortran initial values may also be assigned by using a special form of the type specification statement (see section 10.3.1.8).

Initial values may not be assigned to the dummy arguments of function or subroutine subprograms nor to a variable in a function subprogram whose name is also the name of the subprogram or an entry to the subprogram. If initial values are to be assigned to variables or arrays that form part of a common block (or are equivalenced to items in a common block), this must be done in the block data subprogram (see section 10.3.2) and not in any of the other program units in which the common block is used. Initial values may only be assigned to named common blocks, and not to the blank common block.

Initialisation is carried out when a program unit is loaded, and not when control enters the program unit. Thus when a subroutine or external function is to be entered several times it cannot be assumed that the initial values apply on each occasion of entry.

In the case of character and logical values the type of a value must be the same as that of the variable or array element to which it is being assigned. For arithmetic values, the type of a value should be the same as that of the variable or array element to which it is to be assigned. However, the compiler will perform a conversion between integer and real values if necessary.

In Parallel Fortran, as an extension to the standard language, integer values may be assigned to character variables. See section 10.3.1.6.

Another extension available in Parallel Fortran allows Hollerith constants, character constants and binary, octal, or hexadecimal constants to be used to initialise certain types of variables. See section 10.3.1.2 for Hollerith constants, section 10.3.1.5 for character constants and section 10.3.1.3 for the others.

10.3.1 The DATA Statement

The DATA statement has the form

DATA *nlist*$_1$*/vlist*$_1$*/*, *nlist*$_2$*/vlist*$_2$*/*, ... , *nlist*$_n$*/vlist*$_n$*/*

where:

each *nlist*$_i$ is a list of names of variables, arrays, array elements or substrings, and implied DO lists, separated by commas. The comma after a slash '/' and before a list is optional.

each *vlist*$_i$ is a value list as described in section 10.3.1.1 below.

The values given in each *vlist* are assigned in order to the items given in the corresponding *nlist*. When an array name appears in a *nlist* it is treated as a list of all the elements of the array, in the order given in section 10.1.2.

The number of items listed in each *nlist* (counting each array element of any arrays named as one item) must be the same as the number of initial values given in the associated *vlist*.

10.3.1.1 Value Lists

Values are given in the form of lists of items separated by commas: these items may take either of the following forms:

val
*rept***val*

where *val* is a constant or the symbolic name of a constant, and *rept* is a non-zero unsigned integer constant or the symbolic name of such a constant. The latter form is equivalent to specifying *rept* copies of the constant *val*. Thus the list

```
1,3,4.975,.TRUE.,4*8,3E-2
```

will have the same effect as the list

```
1,3,4.975,.TRUE.,8,8,8,8,3E-2
```

10.3.1.2 Hollerith Constants in Value Lists

In Parallel Fortran you may use Hollerith constants in value lists to initialise variables of any type except character. A Hollerith constant takes the form:

*n*H*text*

where *text* is a string of characters of length *n*. *text* may contain spaces and apostrophes. An example of a valid Hollerith constant is:

```
7H(F10.3)
```

A Hollerith constant used for initialisation must be of the same length as, or shorter than, the variable being initialised.

10.3.1.3 Initialisation with Binary, Octal or Hexadecimal Constants

In Parallel Fortran binary, octal, or hexadecimal constants can be used to initialise logical, integer, or real variables. These constants are indicated by a letter followed by a string in single quotes, where

B	indicates a binary constant (digits 0 and 1)
O	indicates an octal constant (digits 0 to 7)

Z or **X** indicates a hexadecimal constant (digits **0** to **9** and **A** to **F**).

For hexadecimal constants, the string of digits may alternatively precede the letter **X**, instead of following it.

For example, the following constants have the decimal values shown:

Constant	Value
B'0001'	1
O'12'	10
Z'F'	15
X'B' or 'B'X	11

10.3.1.4 Implied-DO in a DATA Statement

The implied-DO list in a **DATA** statement has the form:

(*dlist*, *imp* = p_1,p_2,p_3)

where:

dlist is a list of array element names and implied-DO lists.

imp is the name of an integer variable, known as the implied-DO-variable.

p_1, p_2 and p_3 are integer constant expressions, except that the expressions may contain implied-DO-variables of other implied-DO lists that have this implied-DO list within their ranges.

The range of an implied-DO list is the list *dlist*. An iteration count and the values of the implied-DO-variable are found from p_1, p_2 and p_3 as for a DO-loop (see section 13.3.1), except that the iteration count must be positive. p_3 together with the preceding comma may be omitted.

When an implied-DO list appears in a DATA statement, the list items of *dlist* are specified once for each iteration of the implied-DO list with the appropriate substitution of values for any occurrence of the implied-DO-variable *imp*. The appearance of an implied-DO-variable in a DATA statement does not affect the definition of a variable of the same name elsewhere in the same program unit.

Each subscript expression in the list *dlist* must be an integer constant expression, except that the expression may contain implied-DO-variables of implied-DO lists that have the subscript expression within their ranges. For example:

```
DATA ((A(J,I), I=1,J), J=1,5) / 15*0. /
```

10.3.1.5 Character Values

In standard Fortran, initialisation with character values is restricted to variables and array elements of type character and character substrings. Parallel Fortran allows character values to be used to initialise any variable or array element.

Character values are held as strings of eight-bit characters. One byte of storage will hold one character. If a character constant contains fewer characters than the number required to fill the variable, array element or substring to which it is being assigned as an initial value, space characters will be added to the right-hand end of the string to make the number of characters equal to the length of the variable, array element or substring.

If a character constant contains more characters than the number required to fill the variable, array element or substring to which it is being assigned as an initial value, the surplus rightmost characters in the constant are ignored. For example, the character constant

```
'HEAD'
```

could be assigned as an initial value to a character variable of length 4, and would fill it exactly. If the character constant

```
'ARRAY␣ELEMENTS'
```

were assigned as an initial value to an element of a CHARACTER*8 array, the element would hold 'ARRAY␣EL'.

10.3.1.6 Integer Values in Character Variables

In Parallel Fortran, variables or array elements of type character are not restricted to being assigned character values. As an extension to the ANSI Standard, a single character in a variable or array element may be initialised by any integer in the range 0 to 255.

For example, the value 135 could be assigned to a character variable of length 1.

10.3.1.7 Examples of Initial Value Assignment

The effect of the statement

```
DATA I/1/,J/1,3*0,1,3*2,1/
```

where J is a 3 x 3 array, is to assign to I the initial value 1, and to assign to J the values:

```
1 0 0 0 1 2 2 2 1
```

The effect of the statement

```
DATA A,B,C,D/4*0.0/,HEAD/'VALUES'/
```

is to assign the initial value 0.0 to each of real variables A, B, C and D, and to insert the characters 'VALUES␣␣' in the character*8 variable HEAD.

10.3.1.8 Initialisation in a Type Statement

In Parallel Fortran, initial values may be assigned in a type specification statement (see section 9.3.5), in which case the value assigned

to a variable, or the values assigned to the elements of an array, appear immediately after the name of the variable or array in question, bounded by slashes '/'. Thus the example

```
REAL PI/3.14159/, ARRAY(10)/ 5*0.0, 5*1.0/
```

declares the variable PI and the array ARRAY to be of type real. It also assigns PI the value 3.14159, assigns five of ARRAY's ten elements the value 0.0, and the other five the value 1.0.

10.3.2 Block Data Subprogram

Block data subprograms are used to give initial values to items in named common blocks by means of DATA statements. A block data subprogram must start with a BLOCK DATA statement and end with an END statement and may only contain the following statements:

```
IMPLICIT
PARAMETER
Explicit type specification statements
DIMENSION
COMMON
EQUIVALENCE
DATA
```

A block data subprogram is never executed.

If variables and arrays from a common block are named in a COMMON statement in a block data subprogram, then the total storage area in the common block must be specified completely. For example, an array must have its dimension information specified in the COMMON statement or in a DIMENSION or a type statement in the subprogram, even if it is not named in DATA statements.

If any part of a common block is being given an initial value then a complete set of specification statements for the whole block must be included before any part is initialised. This means that DIMENSION, COMMON, EQUIVALENCE and type specification statements must come before DATA statements for each common block.

A Fortran program may contain more than one block data subprogram but any one common block can be referred to in only one block data subprogram. Initial data values may be entered into more than one common block in a single block data subprogram. Items in the blank common block (see section 10.2.3.1) cannot be given initial values.

The block data subprogram may be given an optional name, in which case the name must not be the same as any local name in the subprogram, nor the same as the name of any external procedure, main program, common block or other block data subprogram in the same executable program. There must not be more than one unnamed block data subprogram in an executable program.

As an example, the following block data subprogram gives initial values to some items of the common blocks CB1 and CB2. All the items in each block are specified completely.

```
BLOCK DATA
REAL B(4)
DOUBLE PRECISION Z(3)
COMPLEX C
COMMON/CB1/C,A,B/CB2/Z,Y
DATA B,Z,C/1.0,1.2,2*1.3,3*7.654321D0,(2.4,3.76)/
END
```

Chapter 11

Expressions

In Fortran, expressions may be used in many different statements in a variety of contexts. There are three kinds of expression: *arithmetic* expressions, *logical* expressions and *character* expressions. Arithmetic expressions have numerical values; logical expressions have logical values; and character expressions have character values: this chapter gives the rules for forming and evaluating these kinds of expression.

11.1 Arithmetic Expressions

An *arithmetic expression* is a sequence of arithmetic elements of type integer, real, double precision, complex or double complex, combined by arithmetic operators and parentheses. The type of an arithmetic expression depends upon the types of its constituents; see section 11.1.6.

A byte variable, when included in an expression, is equivalent to an integer element.

11.1.1 Arithmetic Elements

An *arithmetic element* can be a numerical constant, an arithmetic symbolic constant name, a variable name, an array element reference or a function reference (see section 14.2.1.1). For example, the following are valid arithmetic elements:

```
7E23    VARI    A(1,3)    SIN(X)
```

The simplest arithmetic expression is one that consists of only one arithmetic element: the expression is then of the same type as the element. The term expression is used in this manual to include elements as well as more complicated expressions.

11.1.2 Arithmetic Operators and Parentheses

Arithmetic operators are used to combine arithmetic elements or other arithmetic expressions to give more complex arithmetic expressions. The arithmetic operators are:

Operator	Meaning
+	Addition
-	Subtraction or negation
*	Multiplication
/	Division
**	Exponentiation

Some simple examples of arithmetic expressions using only one operator are:

```
-A
```

```
A+B
```

`A*B` Equivalent to the algebraic expression $a \times b$ or ab

`A**B` Equivalent to the algebraic expression a^b

Parentheses are used to enclose arithmetic expressions which form part of a more complex arithmetic expression. The parenthesized expressions are evaluated as separate entities; this usage of parentheses is therefore equivalent to normal mathematical usage.

11.1.3 Rules

When writing arithmetic expressions, the following rules must be observed:

1. Arithmetic elements must be separated by an arithmetic operator.
2. No two operators may be adjacent.
3. The operators '+' and '-' must be followed by an element: the other operators must be both preceded and followed by elements.

Thus the following are not valid arithmetic expressions:

`A.B`	Rule 1: `A` multiplied by `B` must be written as '`A*B`'
`A**-B`	Rule 2: `A` to the power `-B` must be written as '`A**(-B)`'
`*B`	Rule 3: this on its own is meaningless, while '`-B`' is valid

11.1.4 Order of Evaluation

Arithmetic expressions are evaluated in general from the innermost set of parentheses outwards. Within each set of parentheses or each unparenthesized expression, the order of evaluation is from left to right, except when the precedence of operators dictates otherwise. This precedence is:

1. Function references

2. Exponentiations

3. Multiplications and divisions

4. Additions and subtractions

This order of precedence determines the sequence of operations in the evaluation of an expression. The first two operators are compared and, if the first takes precedence over or is equal to the second, then the first operation is performed. If the second takes precedence over the first, the third operator is compared with the second and so on. When the end of the expression is reached, any remaining operations are performed, reading from right to left. For example, in the expression `A*B+C*D**I` the operations are performed as follows:

1.	`A*B`	⇒`E1`	intermediate result	`E1+C*D**I`
2.	`D**I`	⇒`E2`	intermediate result	`E1+C*E2`
3.	`C*E2`	⇒`E3`	intermediate result	`E1+E3`
4.	`E1+E3`	⇒*result*	final operation	

If one exponentiation operator follows immediately after another, the evaluation is from right to left. Thus `A**B**C` is evaluated as follows:

1.	`B**C`	⇒`E1`	intermediate result
2.	`A**E1`	⇒*result*	final operation

A series of multiplications and divisions is evaluated from left to right. Under some circumstances this could lead to results that are inaccurate owing to rounding errors or to a lack of precision in the values of the elements in use. If such errors are possible, the programmer may use parentheses to control the order of evaluation so as to produce the most precise result.

Where part of an expression is contained within parentheses, that part is evaluated first, and the result obtained is used in evaluation of the expression as a whole. Where nested parentheses occur, that part of the expression contained within the innermost set is evaluated first.

The sign of a signed quantity takes the same precedence as the addition or subtraction sign. Thus

```
A = -B      is treated as   A = 0 - B
A = -B*C    is treated as   A = -(B*C)
A = -B+C    is treated as   A = (-B) + C
```

11.1.5 Examples of Arithmetic Expressions

The expression

```
ARRAY(2,10)-COS(Z)/(2*PI)
```

is evaluated as follows:

`COS(Z)`	⇒`E1`	function reference
`(2*PI)`	⇒`E2`	parenthesized expression
`E1/E2`	⇒`E3`	division
`ARRAY(2,10)-E3`	⇒*result*	subtraction

The expression

```
A+B*C/D*(P-1)-3.0**(Q+R)+2.0/X**2
```

is evaluated as follows:

```
B*C       ⇒E1
E1/D      ⇒E2
P-1       ⇒E3
E2*E3     ⇒E4
A+E4      ⇒E5
Q+R       ⇒E6
3.0**E6   ⇒E7
E5-E7     ⇒E8
X**2      ⇒E9
2.0/E9    ⇒E10
E8+E10    ⇒result
```

Although the order in which expressions are evaluated in these examples may not be exactly that described, it will be mathematically

equivalent. However, the order of evaluation implied by the presence of parentheses will be followed.

11.1.6 Determination of the Type of an Expression

The value of an arithmetic expression can be of any of the standard types integer, real, double precision, complex, or, in Parallel Fortran, double complex. The evaluation of an expression is carried out in simple steps (as in the example in section 11.1.5). The types of the elements involved in each step determine the type of the value produced by that step. The type of the final expression can be found by following through the steps of the evaluation, noting the types of the intermediate values at each stage.

Table 11.1 gives the type of an expression composed of two simpler expressions of type A and type B. In standard Fortran, an expression composed of a complex expression and a double precision expression is prohibited. In Parallel Fortran, however, such an expression is allowed. It will be of type double complex, as indicated in table 11.1.

As we noted above, a byte variable, when included in an expression, is equivalent to an integer element. This means that an expression including only byte variables will be of type integer.

11.1.7 Integer Arithmetic

The following special considerations apply when both the arguments of an arithmetic operation are of type integer:

- A result of type integer is that integer obtained by truncating the mathematical result towards zero:

`15/4`	$= 3$	(3.75)
`-15/4`	$= -3$	(-3.75)
`4**(-1)`	$= 0$	(0.25)

- A series of multiplication and division operations on integer quantities will always proceed from left to right.

11.1.8 Arithmetic Constant Expressions

An *arithmetic constant expression* may contain only arithmetic constants and arithmetic symbolic constants. The exponentiation operator is not permitted unless the exponent is of type integer.

Arithmetic constant expressions may be used in **PARAMETER** or **DATA** statements.

11.1.9 Integer Constant Expressions

An *integer constant expression* is an arithmetic constant expression in which each constant or symbolic constant is of type integer.

Integer constant expressions may be used in **PARAMETER** statements, as a character length or array bound specifier in a specification statement or as an array element subscript or character substring position expression in **EQUIVALENCE** or **DATA** statements.

11.1.10 Not-a-Number and Infinity

Arithmetic on real, double precision, complex or double complex constants and variables may result in the special values NaN, $+\infty$ and $-\infty$ (that is, Not-a-Number and Positive and Negative Infinity), in the circumstances defined in the IEEE standard. This does not stop the execution of the program, however, and subsequent processing of these values continues in the way the IEEE standard specifies.

In addition, some of the intrinsic functions return these special values, in circumstances defined for the corresponding Inmos functions by the *Standalone Compiler Implementation Manual*[13].

	Type of A				
Type of B	Integer	Real	Double precision	Complex	*Double complex
Integer	Integer	Real	Double precision	Complex	*Double complex
Real	Real	Real	Double precision	Complex	*Double complex
Double precision	Double precision	Double precision	Double precision	*Double complex	*Double complex
Complex	Complex	Complex	*Double complex	Complex	*Double complex
*Double complex	*Double complex	*Double complex	*Double complex	*Double complex	*Double complex

Table 11.1: Expression Types

Note: * = non-standard type

There is no method for representing NaN, $+\infty$ or $-\infty$ in a Fortran program. See sections 15.3.1.2 and 15.3.1.3 for a description of how these values are output.

11.2 Character Expressions

A character expression is a sequence of one or more character elements separated by character operators.

11.2.1 Character Elements

A *character element* can be a character constant, a character symbolic constant name, a character variable name, a character array element reference, a character substring reference or a character

function reference. For example, the following are valid character elements:

```
'SOME TEXT'
CVAR
NAME(I)
```

provided that CVAR has been specified as of type character and that NAME has been specified as a one-dimensional array of type character, or is a function of type character.

11.2.2 Character Operator and Parentheses

The *concatenation operator*, '//', is a character operator that is used to concatenate two character elements to produce a character string of type character whose length is the sum of the lengths of the two elements. Except in a character assignment statement, a character expression must not concatenate a character element whose length specification is an asterisk in parentheses unless the element is the symbolic name of a constant. Parentheses may have a cosmetic effect on a character expression but they do not affect the value found. For example, the following two expressions:

```
'AN'//('AESTHETIC'//'ALLY')
('AN'//'AESTHETIC')//'ALLY'
```

are equivalent, each producing

```
'ANAESTHETICALLY'
```

11.3 Logical Expressions

A *logical expression* is a sequence of logical elements and relational expressions combined by logical operators and parentheses. The value of a logical expression is always either .TRUE. or .FALSE.

11.3.1 Logical Elements

A *logical element* is a constant, a symbolic constant, a variable, an array element or a function reference of type logical (see section 9.1). The value of a logical element must be either `.TRUE.` or `.FALSE.`. For example, the following are valid logical elements:

```
.TRUE.    LVAR    STATUS(1,3)    OK(B)
```

provided that `LVAR` has been specified as of type logical, that `STATUS` has been specified as a two dimensional array of type logical, and that `OK` is a function of type logical. The simplest logical expression is one that consists of only one logical element.

11.3.2 Relational Expressions

A *relational expression* has the form

expr$_1$ *relop* *expr*$_2$

where:

expr$_1$ and *expr*$_2$
: are both arithmetic expressions or are both character expressions.

relop
: is one of the following *relational operators*:

Operator	Meaning
`.LT.`	Less than
`.LE.`	Less than or equal to
`.EQ.`	Equal to
`.NE.`	Not equal to
`.GT.`	Greater than
`.GE.`	Greater than or equal to

The periods are essential.

A complex or double complex operand is only permitted when the relational operator is .EQ. or .NE.

If $expr_1$ and $expr_2$ are character expressions of different lengths then the shorter operand is considered as if it were extended with blanks on the right to the length of the longer operand.

If the relation indicated by the relational operator between the two arithmetic expressions is true, then the value of the relational expression is .TRUE.. Similarly, if the relation is false, the value of the expression is .FALSE.

A relational expression is equivalent to a logical element for the purpose of constructing further logical expressions; it need not be enclosed in parentheses.

11.3.3 Logical Operators and Parentheses

Logical operators are used to combine logical elements, relational expressions or other logical expressions to give more complex logical expressions. The logical operators are defined as follows, where $expr_1$ and $expr_2$ are logical expressions:

.NOT. $expr_1$ This expression has the value .TRUE. if $expr_1$ has the value .FALSE. and has the value .FALSE. if $expr_1$ has the value .TRUE.

$expr_1$ .AND. $expr_2$
This expression has the value .TRUE. if both $expr_1$ and $expr_2$ have the value .TRUE.. It has the value .FALSE. if either $expr_1$ or $expr_2$ or both have the value .FALSE.

$expr_1$ .OR. $expr_2$
This expression has the value .TRUE. if either $expr_1$ or $expr_2$ or both are .TRUE.. It has the value .FALSE. if both $expr_1$ and $expr_2$ are .FALSE.

expr$_1$ **.EQV.** *expr*$_2$
: This expression has the value **.TRUE.** if a and b both have the same value **.TRUE.** or **.FALSE.**. It has the value **.FALSE.** if *expr*$_1$ and *expr*$_2$ have different truth values.

expr$_1$ **.NEQV.** *expr*$_2$
: This expression has the value **.TRUE.** if *expr*$_1$ and *expr*$_2$ do not both have the same value **.TRUE.** or **.FALSE.**. It has the value **.FALSE.** if *expr*$_1$ and *expr*$_2$ have the same truth value.

expr$_1$ **.XOR.** *expr*$_2$
: This expression is available in Parallel Fortran as an alternative to **.NEQV.**. It has the same values as **.NEQV.** in the same circumstances.

The periods are essential.

Parentheses may be used to enclose logical expressions which form part of more complex logical expressions. Their usage here is analogous to their usage in arithmetic expressions. An expression enclosed in parentheses must satisfy the rules given below.

11.3.4 Rules

When writing logical expressions, the following rules must be observed:

1. If arithmetic expressions appear, they must be in pairs separated by relational operators.

2. Logical elements (and relational expressions) must be separated by logical operators.

3. No two logical operators may be adjacent unless the first is one of **.AND.**, **.OR.**, **.EQV.**, **.NEQV.**, or **.XOR.** and the second is **.NOT.**

4. The logical operator .NOT. must be followed by, but must not be preceded by, a logical element. The logical operators .AND., .OR., .EQV., .NEQV., and .XOR. must be preceded by a logical element and must be followed either by a logical element or by the operator .NOT.

Thus the following are not valid logical expressions:

`A.AND.7.0` (Rule 1: `A.AND.7.0.GT.B` is valid if `B` is an arithmetic element and `A` is of type logical)

`A+C` (Rule 2: `A.AND.C` is valid if `A` and `C` are of the type logical. `A+C` is, of course, a valid arithmetic expression if `A` and `C` are arithmetic elements)

`A.AND..OR.C`
(Rule 3: `A.AND..NOT.C` is valid)

`A.NOT.C` (Rule 4: `.NOT.C.AND.A.OR..NOT.D` illustrates the various possible valid uses of operators)

11.3.5 Order of Evaluation

Logical expressions are in general evaluated starting at the innermost set of parentheses and working outwards. Within one set of parentheses or within one expression the order of evaluation is as follows:

1. Evaluation of functions
2. Exponentiation
3. Multiplication and division
4. Addition and subtraction
5. Relational expressions
6. .NOT. operations

7. .AND. operations

8. .OR. operations

9. .EQV., .NEQV., or .XOR. operations

Rounding errors (see section 11.1.4) cannot occur with logical expressions since these may only take the values .TRUE. or .FALSE..

No more code than necessary is actually obeyed when evaluating an expression. Some parts of a logical expression may not be evaluated every time the expression occurs. In the following example

```
A.OR.LGF(.TRUE.)
```

the LGF function need not be called to evaluate the expression when A has the value .TRUE.

11.3.6 Examples of Relational and Logical Expressions

The expression

```
B**2.GT.4.*A*C
```

is a relational expression. The arithmetic expressions are evaluated as described in section 11.1.4. Then the relational operation is performed. The resulting value will be .TRUE. or .FALSE.

The expression

```
JOB.EQ.3.AND.AGE.LT.18
```

is evaluated as follows:

JOB.EQ.3	⇒E1
AGE.LT.18	⇒E2
E1.AND.E2	⇒*result*

The expression

```
A.OR.B.OR.C.AND.(P.OR.Q).AND.(I.LT.1.OR.J.EQ.0)
```

is effectively evaluated as follows:

```
A.OR.B              ⇒E1
P.OR.Q              ⇒E2
C.AND.E2            ⇒E3
I.LT.1              ⇒E4
J.EQ.0              ⇒E5
E4.OR.E5            ⇒E6
E3.AND.E6           ⇒E7
E1.OR.E7            ⇒result
```

However, certain stages are sometimes unnecessary. For example in the above expression, if **A** is **.TRUE.** then the result is **.TRUE.**, regardless of the values of the other variables. Similarly, if **A**, **B** and **C** are all **.FALSE.**, the result will always be **.FALSE.** too.

The order of evaluation in these examples may not be exactly as shown, but will be logically equivalent.

Chapter 12

Assignment Statements

Assignment statements are used in Fortran to assign new values to variables or array elements, replacing any existing values. Assignments are executable statements. There are three types of assignment statements: arithmetic assignment, logical assignment and character assignment. The rules for forming assignment statements are given in this chapter.

12.1 Arithmetic Assignment Statements

An *arithmetic assignment statement* is used to assign a new value to a variable or an array element of type integer, real, double precision, complex, or in Parallel Fortran, byte or double complex. The statement takes the form:

name = *expression*

where:

name is the name of the variable or array element of type integer, real, double precision, complex, or double complex.

expression is an arithmetic expression (see section 11.1).

When an assignment statement is encountered in a program the expression part is evaluated and the resulting value is assigned to the variable or array element with the name *name*. The variable or array element then retains this value until it is assigned a new value. Any previous value of the variable or array element is lost.

The variable or array element name need not be of the same type as the expression. The expression is evaluated according to the rules given in section 11.1 and the resulting value is assigned to the variable or array element after any necessary type transformations as indicated in table 12.1.

Some examples of arithmetic assignment statements follow:

```
VALUE = 38.765
COUNT = COUNT1
NEXT = ITEM(3,1)
MATRIX(7) = MATRIX(4*I) + MATRIX(3)
M = (2*M-1)/Q-M
```

12.2 Logical Assignment Statements

A *logical assignment statement* assigns a new value to a variable or array element of type logical. The statement takes the form:

```
name = expression
```

where:

name is the name of a logical variable or array element.

expression is a logical expression (see section 11.3).

When a logical assignment statement is encountered in a program, the logical expression is evaluated and the resulting value is assigned to the variable or array element with the name *name*. The variable

or array element then retains this value until it is assigned a new value. Any previous value of the variable or array element is lost.

Some examples of logical assignment statements follow:

```
COURSE = .TRUE.
QUAD = B**2.GT.4*A*C
STATUS = PUPIL.AND.AGE.LT.21
RES = A.AND.(B.OR.(C.AND..NOT.D))
```

12.3 Character Assignment Statements

A *character assignment statement* assigns a new value to a variable, substring or array element of type character. The statement takes the form:

name = expression

where:

name is the name of a character variable, substring or array element.

expression is a character expression (see section 11.2).

Note that no character position in *name* may be referenced in *expression*.

The variable, substring or array element need not be of the same length as the character expression. If the expression is shorter, then spaces are added to the right on assignment; but if the expression is longer, then truncation will occur on the right. An example of a character assignment statement is

```
CH = 'TEST DATA'
```

Type of Name	Type of Expression		
	Integer	Real or Double precision	Complex or *Double complex
Integer	Assign	Fix and assign	Fix and assign real part
*Byte	Truncate and assign	Fix, truncate and assign	Fix, truncate and assign real part
Real or Double precision	Float and assign	Assign	Assign real part
Complex or *Double Complex	Float and assign real part, zero imaginary part	Assign real part, zero imaginary part	Assign

Table 12.1: Assignment of Value to Variable or Array Element

* = Non-standard

- *Truncate* means extract the low-order eight bits of the integer value. Should the high-order bit of these eight bits be set, the value, once stored in the byte variable, will be negative.
- *Float* means convert the result to type real.
- *Fix* means truncate any fractional part towards zero and convert the remaining value to integer. There will be an overflow if the result is outside the range of integer values.
- Where real and double precision numbers of different lengths are involved, as much precision is preserved as possible. If a double precision value is assigned to a real variable or array element name, then the value is truncated as necessary. If a real value is assigned to a double precision variable or array element name, then the mantissa is extended with zeros.

Chapter 13

Control Statements

Execution of a Fortran program begins at the first executable statement of the main program. Subsequent statements are executed in the order in which they occur until a control statement is encountered. Control statements are used to transfer control from one part of a program to another. This chapter describes those statements used for transferring control within a program unit. Statements used to transfer control from one program unit to another are described in chapter 14.

13.1 GO TO Statements

A GO TO statement is an executable statement that is used to transfer control to another executable statement in the same program unit. There are three types of GO TO statements: unconditional GO TO, computed GO TO and assigned GO TO.

13.1.1 Unconditional GO TO

An unconditional GO TO statement has the form:

```
GO TO label
```

where *label* is the label of the executable statement to which control is to be transferred (see section 8.3.3).

Each time a GO TO statement of this form is encountered, control is transferred to the statement with the label *label*. This statement must be in the same program unit as the GO TO statement. The first executable statement after the GO TO statement should be labelled unless it is an ELSE IF, ELSE or END IF statement, otherwise control can never reach it.

The following is an example of an unconditional GO TO statement:

```
GO TO 3
```

13.1.2 Computed GO TO

A computed GO TO statement transfers control to one of a list of statements, depending upon the computed value of an expression. The statement has the form:

```
GO TO (label1,label2,...,labeln), iexpr
```

where:

each $label_i$ is the label of an executable statement in the same program unit as the GO TO statement.

iexpr is an integer expression: the preceding comma is optional.

When a computed GO TO statement is encountered *int* should have a value in the range 1 to n. If *int* has the value j, then control is passed to the executable statement with label $label_j$. The same label may appear more than once in the list.

If *int* is not in the required range, then the next statement in sequence will be executed, as the GO TO statement has no effect.

An example of the use of a computed GO TO statement follows:

```
COUNT = 3
⋮
GO TO(14,21,20,15,20),COUNT
```

Control is transferred to the statement with label 20, the third label in the list.

13.1.3 Assigned GO TO and ASSIGN Statements

An assigned GO TO statement transfers control to one of a list of labelled executable statements depending on the value assigned, by an ASSIGN statement, to an integer variable. The statement has one of these forms:

```
GO TO int,(label1,label2,...,labeln)
GO TO int
```

where:

int is an integer variable.

each $label_i$ is the label of an executable statement in the same program unit as the assigned GO TO statement. If present, the bracketed list of labels must contain all those labels that may be assigned to the variable.

The ASSIGN statement has the form:

```
ASSIGN label TO int
```

where:

label is the label of an executable statement in the same program unit as the ASSIGN statement.

int is the integer variable to be used in an assigned GO TO statement.

Each time an assigned **GO TO** statement is encountered, control is transferred to the statement with label *label*, where *label* is the value last assigned to the variable *int* in an **ASSIGN** statement. If, when the statement is encountered, the variable *int* has not been assigned a value by an **ASSIGN** statement in the same program unit, then the effect of the **GO TO** statement is unpredictable.

A variable may have, at different times, a statement label value assigned by an **ASSIGN** statement or an integer value assigned in any other way. An attempt to use a variable which has a statement label value when an integer value is required, or vice versa, will have an unpredictable effect. The label list when present enables the compiler potentially to check that an appropriate label has been assigned.

For example, the statements

```
ASSIGN 57 TO MEANS
:
GO TO MEANS,(92,3,9999,57)
```

will result in a transfer of control from the **GO TO** statement to the statement with label 57. If the **GO TO** statement were written

```
GO TO MEANS
```

the same result would be achieved.

13.2 IF Statements

An **IF** statement allows the program to take different actions depending on a particular condition. Thus an **IF** statement may have one result the first time it is executed in a program and a different result on a subsequent execution if the relevant condition has altered. There are three types of **IF** statement: arithmetic **IF**, logical **IF** and block **IF**.

13.2.1 Arithmetic IF

An arithmetic IF statement transfers control to one of three statements depending on the value of an arithmetic expression. It has the form:

IF (*expr*) $label_1$,$label_2$,$label_3$

where:

expr is an arithmetic expression of type integer, real or double precision.

$label_1$,$label_2$,$label_3$
are the labels of executable statements in the same program unit as the IF statement. The same label may be used more than once.

The statement causes control to be transferred to the statement with label $label_1$, $label_2$ or $label_3$ depending on whether the value of the expression is less than, equal to, or greater than zero respectively.

For example, the statement

```
IF(B*B-4.0*A*C) 100,101,102
```

would have the following effects:

- If $B^2 - 4 \times A \times C < 0$, control is transferred to the statement with label 100.
- If $B^2 = 4 \times A \times C$, control is transferred to the statement with label 101.
- If $B^2 - 4 \times A \times B > 0$, control is transferred to the statement with label 102.

Note: As real expressions rarely evaluate exactly to zero, programs should assume that the branch to $label_2$ will never be taken when *expr* is real.

13.2.2 Logical IF

The logical IF statement tests whether a logical expression has the value .TRUE. or .FALSE.; if it has the value .TRUE. then a particular executable statement included in the IF statement is executed. The statement has the form:

```
IF ( expr ) statement
```

where:

expr is a logical expression.

statement is any executable statement except a DO, block IF, ELSE, ELSE IF, END IF, or END statement or another logical IF statement.

If *expr*, when evaluated, has the value .TRUE., *statement* is executed; if *expr*, when evaluated, has the value .FALSE., *statement* is not executed. For example, if the statement

```
IF(B*B.LT.4.0*A*C) GO TO 100
```

is executed when $B^2 < 4 \times A \times C$, control is transferred to the statement with label 100. Otherwise the GO TO statement is ignored.

Some other examples of logical IF statements follow:

```
IF(SUM+TERM.GE.9E7.OR.TERM.LT.1E-2) CALL CHECK
IF(COUNT.EQ.60) IF(COUNT1-14)26,27,28
IF(I.LT.0) I = -1
```

13.2.3 Block IF

A block IF statement is used with the END IF statement and optionally with the ELSE and ELSE IF statements to control the execution of a block of consecutive executable statements. The block of statements is divided into IF-blocks, ELSE-blocks and ELSE IF-blocks. The block IF statement and IF-blocks, the ELSE statement

and ELSE-blocks, and the ELSE IF statement and ELSE IF-blocks are described in sections 13.2.3.1, 13.2.3.2 and 13.2.3.3 respectively. The END IF statement is described in section 13.2.3.4.

Every statement in an IF-block, an ELSE-block or an ELSE IF-block has an IF-level. The IF-level of a statement s is $n_1 - n_2$, where n_1 is the number of block IF statements from the beginning of the program unit down to and including s, and n_2 is the number of END IF statements in the program unit down to but not including s.

The IF-level of every statement must be non-negative, the IF-level of each block IF, ELSE, ELSE IF and END IF statement must be positive: the IF-level of the END statement must be zero.

Figure 13.1 shows how IF-levels are assigned to statements.

13.2.3.1 The Block IF Statement and IF-blocks

The block IF statement has the form:

```
IF ( expr ) THEN
```

where *expr* is a logical expression.

An IF-block consists of all the executable statements following the block IF statement down to but not including the next ELSE, ELSE IF or END IF statement that has the same IF-level as the block IF statement. An IF-block may be empty.

When the block IF statement is executed, *expr* is evaluated and if it has the value .TRUE. then the IF-block is executed. If the IF-block is empty and the value of expression is .TRUE. then control passes to the next END IF statement at the same IF-level as the block IF statement. If the value of *expr* is .FALSE. then control passes to the next ELSE IF or END IF statement that has the same IF-level as the block IF statement.

Control cannot be passed into an IF-block. If the last statement in an IF-block does not pass control elsewhere then control passes to

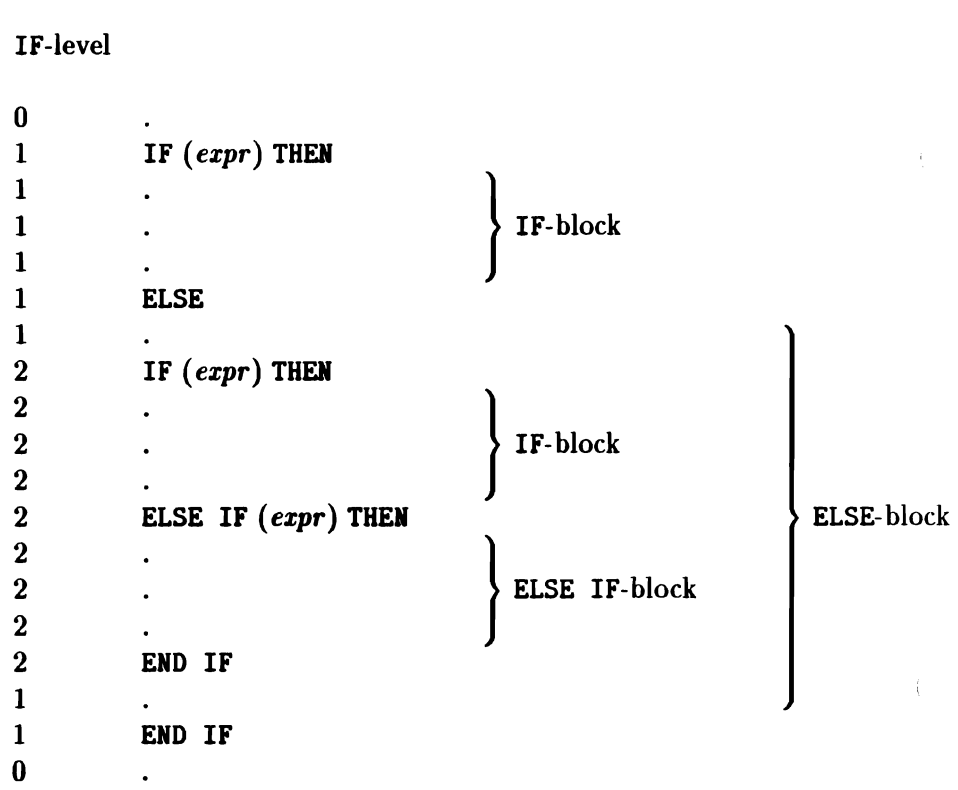


Figure 13.1: IF-levels of statements

the next **END IF** statement that has the same IF-level as the block IF statement preceding the IF-block.

13.2.3.2 The ELSE Statement and ELSE-blocks

The ELSE statement has the form:

```
ELSE
```

An ELSE-block consists of all the executable statements following the ELSE statement down to but not including the next **END IF** statement that has the same IF-level as the ELSE statement. An ELSE-block may be empty.

An **END IF** statement of the same IF-level as the **ELSE** statement must be included before an **ELSE IF** or another **ELSE** statement of the same IF-level.

Control cannot be passed into an **ELSE**-block. No reference may be made to the statement label, if any, of the **ELSE** statement.

13.2.3.3 The ELSE IF Statement and ELSE IF-blocks

The **ELSE IF** statement has the form:

```
ELSE IF ( expr ) THEN
```

where *expr* is a logical expression.

An **ELSE IF**-block consists of all the executable statements following the **ELSE IF** statement down to but not including the next **ELSE**, **ELSE IF** or **END IF** statement that has the same IF-level as the **ELSE IF** statement. An **ELSE IF**-block may be empty.

When the **ELSE IF** statement is executed, *expr* is evaluated and if it has the value **.TRUE.** the **ELSE IF**-block is executed. If the **ELSE IF**-block is empty and *expr* has the value **.TRUE.** then control passes to the next **END IF** statement that has the same IF-level as the **ELSE IF** statement. If the value of *expr* is **.FALSE.** then control passes to the next **ELSE**, **ELSE IF** or **END IF** statement that has the same IF-level as the **ELSE IF** statement.

Control cannot be passed into an **ELSE IF**-block. No reference may be made to the statement label, if any, of the **ELSE IF** statement.

13.2.3.4 The END IF Statement

The **END IF** statement has the form:

```
END IF
```

Each block IF statement must be matched by a separate END IF statement.

13.3 DO Loops

A DO-loop is a series of statements that is to be executed several times. It is headed by a DO statement which specifies the number of times the loop is to be executed and also specifies the last statement included in the loop. The range of a DO-loop is the series of statements from the statement after the DO statement down to and including the terminal statement. If a DO statement appears within an IF-block, ELSE-block or ELSE IF-block, then the range of the DO-loop must be wholly within that block.

13.3.1 DO Statements

The DO statement has the form:

DO *label* *int* = p_1,p_2,p_3

where:

label is the label of the terminal statement (see section 13.3.3 below for further details).

int is the DO-variable.

p_1 is the initial parameter.

p_2 is the terminal parameter.

p_3 is the incrementation parameter and may be omitted, together with the preceding comma.

As an extension in Parallel Fortran, a DO-loop may terminate with an END DO statement. In this case the label is omitted from the DO statement.

The terminal statement must be in the same program unit as the DO statement and must occur later in the program unit than the DO statement. The DO-variable must be an integer, real or double precision variable. The three parameters must all be integer, real or double precision expressions. The incrementation parameter must not be zero at the time of execution of the DO-loop. If the incrementation parameter is omitted, it is assumed to have a value of 1.

When a DO statement is encountered in a program, the values of the three parameters are calculated and, if necessary, converted according to the rules given in section 12.1 to be of the same type as the DO-variable.

The value of p_1 is assigned to the DO-variable and the iteration count is found by evaluating this expression:

MAX(INT((p_2 - p_1 + p_3)/p_3), 0)

If this count is non-zero then execution of the first statement in the range of the DO-loop begins. When the terminal statement is reached the DO-variable is incremented by p_3. The iteration count is decremented by one. If it is non-zero, control then returns to the first statement in the range of the DO-loop. If it is zero, the DO-statement is satisfied: the DO-variable retains its current value and control passes to the next executable statement following the terminal statement.

Variables and array elements used in the expressions for the parameters p_1, p_2 and p_3, and the DO-variable *int*, may be referenced within a DO-loop, but the value of *int* must not be altered by any statement within the range of the DO statement. For example, in the series of statements

```
      SUMSQ = 0.0
      SUM = 0.0
      DO 27 J = 1,MAX,2
      SUM = SUM + PART(J+1)
27    SUMSQ = SUMSQ + PART(J)*PART(J)
```

the range of the DO statement is the two statements following it. The effect of the sequence is to set SUM equal to PART(2)+PART(4)+...+PART(x+1), where x is the greatest odd integer less than or equal to MAX, and to set SUMSQ equal to the sum of the squares of PART(1),PART(3),...,PART(

Control may be transferred out of a DO-loop before the DO statement is satisfied (for example, by use of a GO TO statement): in this case the control variable retains its current value (see section 13.3.5). Transfer into the range of a DO-loop from outside the range is forbidden in standard Fortran 77. However, Parallel Fortran allows extended range DO-loops as an extension to the language. A control statement may be used to transfer control out of the range of the loop. A subsequent control statement may then transfer control back into the range of the same DO-loop. The value of the DO variable must not be changed outside the range of the DO-loop.

Execution of a function reference or a CALL statement that appears within the range of a DO-loop is permitted. If control is returned from the subprogram by means of an extended form of the RETURN statement (see section 14.2.2.2) to a statement not in the range of the DO-loop, then control cannot be transferred into the range of the DO-loop.

13.3.2 The DO WHILE Statement

DO WHILE is an alternative form of the DO statement available under Parallel Fortran. It has the form:

```
DO label WHILE ( expr )
```

where:

label is the label of the terminal statement.

expr is a logical expression.

The DO-loop is repeated for as long as *expr* has the value .TRUE.. For example,

```
      X = 0
      Y = 121
      DO 1001 WHILE (X.LE.Y)
      :
1001  X = X + 2
```

would execute the DO WHILE loop 61 times, until X was greater than Y. The value of the logical expression is checked on the first pass as well as on later ones, so that if expression has the value .FALSE. to start with then the DO WHILE-loop would not be executed at all. If no label is included, the DO WHILE loop must end with an END DO statement.

13.3.3 Terminal Statements

The range of a DO-loop consists of all the executable statements following a DO statement up to and including the terminal statement which is identified by the label in the DO statement. More than one DO-loop may share the same terminal statement.

The terminal statement must not be any of the following statements:

- Unconditional GO TO
- Assigned GO TO
- RETURN
- STOP
- DO
- Arithmetic IF
- Block IF
- ELSE
- ELSE IF

- **END IF**
- **END**
- Logical **IF** containing any one of the following:
  - **DO**
  - Block **IF**
  - **ELSE**
  - **ELSE IF**
  - **END IF**
  - **END**
  - Another logical **IF**

A labelled **CONTINUE** statement may be used as the terminal statement to overcome this restriction (see section 13.4).

In Parallel Fortran, an **END DO** statement may be used instead of a labelled statement to terminate a DO-loop. Thus

```
SUMSQ = 0.0
SUM = 0.0
DO J = 1,MAX,2
SUM = SUM + PART(J+1)
SUMSQ = SUMSQ + PART(J)*PART(J)
END DO
```

would have the same effect as in the example in section 13.3.1.

13.3.4 Nested DO-Loops

The statements included in the range of a DO statement may include other DO statements; the DO-loops are then said to be nested. In a system of nested DO-loops, the range of any inner DO statement must be completely contained in the range of any outer DO statements. However, DO statements may share a terminal statement. If two or

more DO statements share the same terminal statement then the DO-variable for any outer DO is not increased and tested until all inner DOs are satisfied.

For example the sequence of statements

```
      DIMENSION A(10,10),B(10,10),C(10,10)
      DO 20 J = 1,10
      DO 20 I = 1,10
20    C(I,J) = A(I,J)+B(I,J)
```

forms the elements of array C by adding together the corresponding elements of arrays A and B. When the first DO statement is encountered J is set equal to 1. Then the second DO statement is encountered, and the inner DO-loop is performed with J equal to 1 and I varying from 1 up to 10. Only after that is J increased to 2.

13.3.5 Transfer of Control in DO-Loops

Any transfers of control may occur within the range of a DO statement except that, if a statement is the terminal statement for more than one DO statement, then control can be transferred to it only from the range of the innermost DO having that terminal statement. A CONTINUE statement (see section 13.4) may be used to overcome this restriction.

Any transfers of control from inside to outside a DO-loop are allowed. The DO-variable retains its current value.

For example, the sequence of statements

```
      SUM = 0.0
      DO 25 J = 1,100
      IF (A(J).GT.50)GO TO 80
25    SUM = SUM + A(J)
      ⋮
80    statement
```

forms the sum of the elements of an array, except that if any element is greater than 50, control passes to statement 80 outside the DO-loop.

13.4 The CONTINUE Statement

The **CONTINUE** statement is a dummy statement and causes no action. It has the form

```
CONTINUE
```

This statement is most often used to give distinct terminal statements to a nest of DO-loops. It is also useful for avoiding the statements forbidden in section 13.3.3.

For example, in the sequence of statements

```
      X = 0
      DO 200 I = 5, 100, 5
      Y = A(I,1)
      IF(Y.LT.0.0)GO TO 200
      X = X + Y
      DO 300 J = 2, 150
300   X = X + A(I,J)
200   CONTINUE
```

the **CONTINUE** statement is necessary to allow the **IF** statement to transfer control to the terminal statement of the outer loop without also transferring control to the terminal statement of the inner loop.

13.5 STOP Statements

A **STOP** statement terminates the execution of the program. It has one of the following forms:

```
STOP
STOP n
STOP 'message'
```

where:

n is a string of one to five digits.

message is a literal constant enclosed in apostrophes.

When a STOP statement is encountered in a program, no further statements are executed, and the run is terminated. If the second or third varieties of STOP is used, a message in one of the following formats is output:

```
STOP n
STOP message
```

Important Notes

- The STOP statement should not be executed in a subroutine which has been invoked in a subsidiary thread, via F77_THREAD_START or F77_THREAD_CREATE. Any attempt to do so may result in a condition handling error (see section 17.6.4.4). In these conditions, use the F77_THREAD_STOP subroutine instead, to stop the current thread only.

- STOP counts as an input-output statement. This means that when there is more than one thread running in a task, a thread must gain control of the run-time library before executing a STOP, by calling the F77_THREAD_USE_RTL subroutine (see section 18.2.3).

13.6 PAUSE Statements

A PAUSE statement causes the program to output a message, and then continue. The statement has one of the following forms:

```
PAUSE
PAUSE n
PAUSE 'message'
```

where:

n is a string of 1 to 5 decimal digits.

message is a literal constant enclosed in apostrophes.

Execution of this statement causes a message of one of the following forms to be reported to the user:

```
PAUSE
PAUSE n
PAUSE message
```

After this, the following message will be output:

```
Type GO to resume execution, any other input will terminate
the job
```

The user may then decide whether to terminate the run or to go back to executing the program.

Important Notes

- The PAUSE statement should not be executed in a subroutine which has been invoked in a subsidiary thread, via F77_THREAD_START or F77_THREAD_CREATE. This is because if the user terminates the job by typing something other than GO, there will be a condition handling error (see section 17.6.4.4).

- PAUSE counts as an input-output statement. This means that when there is more than one thread running in a task, a thread must gain control of the run-time library before executing a PAUSE, by calling the F77_THREAD_USE_RTL subroutine (see section 18.2.3).

- If you attempt to link a program which uses the PAUSE statement against one of the stand-alone run-time libraries safrtlt4.bin or safrtlt8.bin (see section 5.3), the linker will report an error.

Chapter 14

Program Units and the Transfer of Control

As described in chapter 8 a program is made up of *program units*: one *main program* and, possibly, other program units called *subprograms*. The *block data* subprogram, described in section 10.3.2, contains only non-executable statements and control never enters it. The remaining subprograms, called *procedures*, are described below in section 14.1. Section 14.2 describes the transfer of control between program units, involving the use of **CALL** and **RETURN** statements and function references. The following sections describe the transfer of values between program units, the correspondence between dummy and actual arguments of procedures (see also section 14.3) and the facility of multiple entry into a subprogram.

14.1 Procedures

Procedures normally contain sequences of statements that carry out a process that is likely to be repeated in the execution of the program. It is convenient for the programmer to write out such a sequence only

once, so the programmer writes a procedure using dummy arguments and declares it in the program once in that form.

A *dummy argument* is a name that is used in a procedure at the declaration stage. Dummy arguments represent the values that will be associated with the procedure when it is actually called later in the program. At each call of the procedure, the values required in that particular call are substituted for the dummy arguments; these substituted values are called the *actual arguments.*

Besides avoiding the need to write out repeated processes each time, procedures also form logical subdivisions of the program. These subdivisions may be written and tested quite separately from the main body of the program if desired.

The location of the procedure declaration, the scope of the validity of the use of the procedure and how to call the procedure depend on the kind of procedure. There are four kinds of procedure:

1. Intrinsic function
2. External function
3. Statement function
4. External subroutine

Each of these is described in detail in sections 14.1.2 and 14.1.3.

14.1.1 Differences between Function and Subroutine Subprograms

A function subprogram is used to evaluate a specific function and to substitute a value for the function reference in the calling program unit. This class of subprogram would not normally be used to change the value of any variables or array elements in the referencing program unit though facilities are available for the programmer to do so if desired. A subroutine subprogram is not specifically used

for the calculation of a single value but may perform any series of operations. From this basic difference some others follow:

- Function subprograms are entered by function references; subroutine subprograms are entered by `CALL` statements.
- Function subprograms have a type; subroutine subprograms do not.
- Results from a function subprogram are returned principally via the function value; results from subroutine subprograms are returned only via dummy arguments and common blocks.

14.1.2 Functions

14.1.2.1 Intrinsic Functions

The Fortran compiler provides a number of standard functions. These are called *intrinsic functions* and include certain standard mathematical functions, such as sine and cosine, and type conversion functions.

Use of a function name in an `EXTERNAL` statement (see section 14.3.4), will mean that the name will not be recognised as an intrinsic function name.

Intrinsic function names are either *generic* or *specific*. Generic function names provide an automatic function selection facility. This facility allows the programmer to use a single generic name when requesting a Fortran-supplied function which has several specific names, depending on argument type. The proper function is selected by the Fortran Compiler, based on the type of the arguments of the function. With this facility the programmer can, for example, use the generic name `SIN` to refer to any sine routine, rather than explicitly calling `SIN` for real arguments, `DSIN` for double precision arguments, `CSIN` for complex arguments or `CDSIN` for double complex arguments.

Generic function names may have specific function names associated with them. If a specific name is used, then the arguments must be of the correct type otherwise a compile time fault will be reported. Some functions do not have a generic name. The specific names that identify the intrinsic functions, their generic names, function definitions, types of arguments and types of results are given in appendix E.

A specific name of an intrinsic function that appears in an INTRINSIC statement (see section 14.3.5) may be passed as an actual argument to an external procedure with the exception of intrinsic functions for type conversion, lexical comparison, and maximum/minimum functions.

An intrinsic function can be called at any point in any program by means of a function reference (see section 14.2.1.1), in which the function name is given and the dummy arguments are replaced by actual arguments.

Actual arguments can be any expressions of the correct type and therefore may contain function references to other function subprograms (see the fourth example below).

Examples

```
SIN(ANGLEA)
A + SQRT(B)
A**2 + 2.0* COS(BETA + PI/2.0)
A + SIN(ALOG(A+B+C)**3 + SQRT(Z+SQRT(Y)))
```

14.1.2.2 External Functions

Any functions required in a particular program that are not intrinsic functions can be written as *external functions* for that program. External functions are independently written subprograms that are executed whenever a function reference (see section 14.2.1.1) to their name is encountered in any program unit.

An external function is identified as such by the first statement being a function declaration statement. This statement has the form:

type **FUNCTION** *fname* $(x_1, x_2, \ldots, x_n)$

where:

type is an optional type specification, that is, one of the following:

INTEGER
BYTE (in Parallel Fortran only)
REAL
DOUBLE PRECISION
COMPLEX
DOUBLE COMPLEX (in Parallel Fortran only)
LOGICAL
CHARACTER

In the absence of a type specification the function will be given its type by the predefined convention (see section 9.3.1).

The type specification may optionally be followed by a length specification, which may be any of the forms described in sections 9.3.5.1 and 9.3.5.2.

fname is the name of the function being declared; the name by which it will be called elsewhere in the program by a function reference.

$x_1, x_2, \ldots, x_n$
is a dummy argument list which may be empty, although the enclosing brackets must be specified. The items represent variable, array or dummy procedure names.

The **FUNCTION** statement is followed by the statements that make up the required process. The function declaration itself finishes with an **END** statement (see section 8.3.2).

A function can be invoked as a variable anywhere in the program by a function reference (see section 14.2.1.1) giving its name, *fname*, with actual arguments $(a_1,a_2,\ldots,a_n)$. Within the function subprogram, the function name can be used as a variable (of the specified type) and must be assigned a value at least once before a **RETURN** or **END** statement is executed.

A function must not reference itself: recursion, either direct or indirect, is not allowed.

Example of a function subprogram

```
      LOGICAL FUNCTION TMEAN(A,D)
      REAL MEAN,A(10, 10)
      MEAN = 0
      DO 4 I = 1, 10
      DO 4 J = 1, 10
4     MEAN = MEAN + A(I,J)
      MEAN = MEAN/100
      TMEAN = MEAN.GT.D
      END
```

14.1.2.3 Statement Functions

If a mathematical function can be written in one statement then it may be written as a *statement function*. The statement function is declared by a statement function statement which takes the form:

$$sfname(x_1,x_2,\ldots,x_n) = expr$$

where:

sfname is the name given to the function.

$x_1,x_2,\ldots,x_n$ is a dummy argument list which may be empty, although the enclosing brackets must be specified. The names used as dummy arguments in this list may be

used elsewhere in the same subprogram as variables of the same type.

expr is an arithmetic, logical or character expression. It may contain references to external functions or previously defined statement functions. It may be a logical expression only if *sfname* is defined as type logical and a character expression only if *sfname* is defined as type character.

All statement function declarations must precede the first executable statement of the program unit and the statement function can be invoked only within that program unit.

A statement function is invoked in an expression by using *sfname*, with actual arguments replacing the dummy arguments. The actual arguments must correspond in number and type to the dummy arguments.

Example

```
C Declaration of the statement function VOL
      VOL(R,H) = 3.14*R**2*H
C Executable statements
      VOL SUM = 0.
      :
      DO 11 I = 1,15
      READ(5,12) D,X
C This statement includes a reference to VOL with actual
C arguments 0.5*D replacing R and X replacing H
11    SUM = SUM + VOL(0.5*D,X)
```

14.1.3 Subroutines

A subroutine subprogram fulfils a similar purpose to a function subprogram but returns any results in a different way.

A subroutine is declared by a SUBROUTINE statement, which takes the form:

SUBROUTINE *sname* ($x_1, x_2, \ldots, x_n$)

where:

sname is the name of the subroutine.

$x_1, x_2, \ldots, x_n$
is a dummy argument list. The list may be empty, in which case enclosing brackets may be omitted. The list items can represent variable, array or dummy procedure names or can take the form '*'. The character '*' represents a label in the calling program unit (see section 14.2.2.2).

The **SUBROUTINE** statement is followed by the statements that carry out the desired processes and the subroutine declaration is finished by an **END** statement (see section 8.3.2).

The subroutine can be called from another program unit by use of the **CALL** statement (see section 14.2.2.1). Control is returned to this position when the **RETURN** or **END** statement of the subroutine is encountered.

A subroutine must not contain a call to itself; recursion, either direct or indirect, is not allowed.

14.2 Transfer of Control between Program Units

Every executable Fortran program contains at least one program unit, the main program. The first statement of the main program may be a **PROGRAM** statement. This statement has the form:

PROGRAM *progname*

where *progname* is the name of the program and must be of the form specified in section 8.4.

Execution of the program begins at the first executable statement of the main program. Control stays in the main program until either a function reference or a **CALL** statement is encountered, when control will be transferred to another program unit: a function from a function reference or a subroutine from a **CALL** statement. Control will then be passed between the main program and the other program units and between the program units themselves until either a **STOP** statement is executed or the **END** statement of the main program is reached.

The ways of entering and leaving function and subroutine subprograms are described in the sections below.

14.2.1 Functions

14.2.1.1 Transfer of Control to a Function Subprogram

Control is passed to a function subprogram by a function reference. This function reference takes the form:

name $(a_1, a_2, \ldots, a_n)$

where:

name is the name of an external function, an intrinsic function or a statement function.

$a_1, a_2, \ldots, a_n$
is a list of actual arguments that replace the dummy arguments $x_1, x_2, \ldots, x_n$ given in the function declaration. They must agree in order, number and type with the dummy arguments. If the referenced function has no parameters, the reference to it must still include a pair of brackets.

When a function reference to an intrinsic function or a statement function is encountered in a program unit, the function is evaluated

using the actual arguments supplied by the function reference. This value is then substituted where the function reference occurs and execution of the program unit continues.

When a function reference to an external function is encountered in a program unit, control enters the function at the first executable statement of the function subprogram. Within the body of a function, the function name must be assigned a value at least once.

External function names must have an associated type. The type of the external function may be declared in the **FUNCTION** statement; otherwise its type is defined by the initial letter of its name, as described in section 9.3.1. In any program unit in which the external function is referenced, the type must be declared in a type specification statement unless the predefined type is correct. The type assumed by the referencing program unit must agree with that defined in the external function. An external function name must not be assigned an initial value in a **DATA** statement.

14.2.1.2 Return of Control from a Function Subprogram

In the case of a statement function or an intrinsic function, control is returned automatically to the position of the function reference in the calling program unit. In the case of external functions, control is returned from the external function to the calling statement when control reaches a **RETURN** or the **END** statement and the function value is returned to the calling program unit.

The **RETURN** statement takes the form:

```
RETURN
```

An extended form of this statement is available for use in a subroutine subprogram only (see section 14.2.2.2).

14.2.1.3 Example of an External Function

The following is an example of an external function showing how the function is referenced in the calling program unit and how control is returned to the statement containing the function reference.

The function **TMEAN** has the value **.TRUE.** if the mean of the elements of a 10 × 10 array is greater than **D**.

```
      LOGICAL FUNCTION TMEAN(A,D)
      REAL MEAN,A(10, 10)
      MEAN = 0
      DO 4 I = 1, 10
      DO 4 J = 1, 10
4     MEAN = MEAN + A(I,J)
      MEAN = MEAN/100
      TMEAN = MEAN.GT.D
      RETURN
      END
```

TMEAN could be referenced by any statement which may contain a logical expression. For example:

```
      IF(TMEAN(ARR,50.0))GO TO 48
```

The effect of this statement is to test whether the mean of the 10 × 10 array **ARR** is greater than 50. If it is, control passes to the statement labelled **48**.

14.2.2 Subroutines

14.2.2.1 Transfer of Control to a Subroutine Subprogram

Subroutine subprograms are called from another program unit by a **CALL** statement statement. The **CALL** statement has the form:

CALL *sname* $(a_1, a_2, \ldots, a_n)$

where:

sname is the name of the subroutine being called.

$a_1, a_2, \ldots, a_n$
is a list of actual arguments. These must agree in order, number and type with the dummy arguments given in the subroutine declaration for *sname*. If the referenced subroutine has no arguments the CALL to it may omit the brackets.

To correspond with a dummy argument of the form '*', an actual argument must take the form

*n

In Parallel Fortran the actual argument may also have the form

&n

In both cases *n* is the label of an executable statement in the calling program unit (see section 14.2.2.2).

When a CALL statement statement referring to a subroutine is executed, control is transferred to that subroutine, which is entered at its first executable statement.

Subroutine names do not have an associated type.

14.2.2.2 Return of Control from a Subroutine Subprogram

Control is returned from a subroutine subprogram to the calling program unit when control reaches a RETURN or the END statement within the subroutine. There may be any number of RETURN statements in a subroutine but only one of these will be executed in any one execution of the subroutine. The RETURN statement statement has two possible forms:

```
RETURN
RETURN expr
```

where *expr* is an integer expression.

The simple form, **RETURN**, has the effect of returning control to the statement following the **CALL** statement in the calling program unit.

The extended form, **RETURN** *expr*, provides a means of returning to any labelled statement in the calling program unit. The expression *expr* has a value, say *n*, and this value denotes that return is to be made to the *n*th statement label in the argument list. If *expr* is less than one or greater than the number of statement labels in the argument list then control is returned to the statement following the **CALL** statement in the calling program unit.

For example, if a **CALL** statement of the form

```
CALL SUB(X,Y,*3,*10,2)
```

is made to a program unit whose first statement is

```
SUBROUTINE SUB(A,B,*,*,C)
```

and this subroutine contains the following **RETURN** statements:

```
RETURN 2
RETURN 1
RETURN
```

then **RETURN** 2 will return control to the statement labelled 10 in the calling program unit, **RETURN** 1 will return control to the statement labelled 3, and **RETURN** will return control to the statement following the **CALL** statement in the calling program unit.

14.2.2.3 Example of a Subroutine Subprogram

The following is an example of a subroutine subprogram showing how it is called and how control is returned to the calling program unit by means of a simple **RETURN** statement.

Subroutine **ADD** is required to add the elements of matrix **I**.

```
SUBROUTINE ADD(I,J)
DIMENSION I(10)
```

```
      J = 0
      DO 2 K = 1,10
2     J = J + I(K)
      RETURN
      END
```

If this subroutine is to be entered, then the **CALL** statement must refer to the subroutine name **ADD**. For example, if it is required to add together the elements of an array **IARR** and to hold the result in **N**, the calling program unit would contain the following statements:

```
      DIMENSION IARR(10)
      ⋮
      CALL ADD(IARR, N)
```

14.3 Correspondence between Dummy and Actual Arguments

A dummy argument (see section 14.1) must be one of the following:

- A dummy variable name.
- A dummy array name.
- A dummy function or subroutine name.
- An asterisk '*', or alternatively in Parallel Fortran, an ampersand '&', for subroutines only, signifying an alternate return to the calling program unit (see section 14.2.2.2).

Actual arguments may be any of the following:

- An expression except a character expression involving concatenation of a character element whose length specification is an asterisk in parentheses (unless it is a symbolic constant). Note that an expression may be a constant or a symbolic constant.
- An array name.

- An intrinsic function name.
- An external function or subroutine name.
- A statement label preceded by an asterisk '*', or alternatively in Parallel Fortran an ampersand '&', for subroutines only (see section 14.2.2.2).

Note that a statement function name cannot be used as an actual argument.

Examples of actual arguments

`FA`	(the name of an intrinsic or external function)
`A`	(an array name)
`A(2,3)`	(an array element; see below)
`Z`	(a variable)
`SIN(Z)`	(an expression comprising a function reference)
`3.16`	(a constant)
`X+4*Y+3/Z`	(an expression)

However, several rules apply for correct correspondence between dummy and actual arguments. These rules are as follows:

- Actual and dummy arguments must correspond in number, order and type.
- If the dummy argument is a function, the actual argument that replaces it must be the name of an intrinsic or external function.
- If the dummy argument is a subroutine, the actual argument that replaces it must be the name of a subroutine subprogram (see section 14.4.2).
- If the dummy argument is an array, the actual argument that replaces it must be an array or an array element.

- If the dummy argument is a variable, the actual argument that replaces it may be a constant, a variable, an array element, or any other expression.
- If an external function or subroutine includes a transfer of control to another subprogram, the dummy arguments of the external function or subroutine may be used as actual arguments in the nested subprogram.

The transfer of values between program units by means of dummy and actual arrays is described in section 14.4.

14.3.1 Use of Constants and Expressions

If a dummy argument is assigned a value within the function or subroutine subprogram and this is used to return a result, then the corresponding actual argument may not be a constant or an expression.

14.3.2 Use of Variables

Dummy variables must not occur in **COMMON**, **DATA**, **EQUIVALENCE**, **PARAMETER** or **INTRINSIC** statements; they may occur in type specification or **DIMENSION** statements (as bounds of dummy arrays). Dummy variables may not be specified in **NAMELIST** statements.

14.3.3 Use of Arrays and Array Elements

If a dummy argument is an array then the array must be declared in the subroutine or function subprogram in which the dummy array is used (see section 10.2.2). The size of each dimension may be given as an integer or as a variable of type integer. Dummy array names must not occur in **COMMON**, **DATA**, **EQUIVALENCE**, **PARAMETER** or **INTRINSIC** statements.

If the actual argument is an array name, it is made available to the called subprogram starting at its first element. That is, if the dummy array has n elements then the first n elements of the actual argument will be used as the n elements of the dummy array.

An array element used as an actual argument may replace a dummy argument that is either a variable or an array. If an array element replaces a variable, only that one element is made available to the called program unit. If an array element replaces an array, the specified actual element is used as the first element of the dummy array, and subsequent elements of the actual array form the remaining elements of the dummy array.

Thus if the actual argument is an array element, say `A(`x`)`, and the dummy array is specified as having n elements then the n elements of array `A` from `A(`x`)` to `A(`$x + n - 1$`)` are used as the n elements of the dummy array.

The actual argument specified must at least be large enough to cover the dummy array completely. That is, if the actual argument is an array, it must have at least as many elements as the dummy array. If the actual argument is an array element, that part of the actual array from and including the element given as the actual argument to the last element must contain at least as many elements as the dummy array.

Care must be taken when arrays having more than one dimension are used because of the order in which array elements are stored (see section 10.1.2).

An example of the use of arrays and array elements as actual arguments follows:

```
      FUNCTION FUN1(A,B)
      REAL A(500),B(10)
      DO 4 I=1, 500
      :
4     CONTINUE
      DO 5 J=1,10
```

```
      :
5     CONTINUE
      FUN1= ...
      RETURN
      END
```

A reference to this external function could be included in an expression as follows:

```
      REAL LIST(540)
      :
      X=4*FUN1(LIST,LIST(531))
```

In this case, the first 500 elements of array LIST will be used as array A in the function and the elements LIST(531) to LIST(540) will be used as array B.

14.3.3.1 Adjustable Arrays

A dummy array declared using one or more integer variables to define the bounds of the subscripts is an adjustable array. This method of declaration is only permissible for dummy arrays, but it allows the dummy array to have dimensions of different size each time the subprogram is executed, though the number of dimensions must remain constant. Each integer variable which specifies a dimension of an adjustable array must appear either in a common block or as a dummy argument in every dummy argument list which contains the array name.

For example, the subroutine ADD given in section 14.2.2.3 could be rewritten to add together the elements of an array of variable size.

```
      SUBROUTINE ADD(I,J,L)
      DIMENSION I(L)
      J=0
      DO 2 K=1,L
2     J=J+I(K)
      RETURN
      END
```

When the subroutine is called, the CALL statement will specify the size of the actual array to be used. For example

```
CALL ADD(IARR,N,20)
```

would add the elements of a one dimensional array IARR with twenty elements.

14.3.3.2 Assumed Size Arrays

An assumed size array is a dummy array declared with an asterisk as the upper bound of its last dimension (see section 10.2.2). A dummy array declared with assumed size takes its actual size from the corresponding actual argument as follows:

- If the actual argument is a non-character array, the size of the dummy array is the size of the actual argument array.
- If the actual argument is a non-character array element, the size of the dummy array is the size of the actual argument array from the specified array element to the end of the array.
- If the actual argument is a character array, character array element, or character array element substring, the size of the dummy array is the number of characters from the specified actual argument to the end of the actual argument array, divided by the length of an element in the dummy array.

Note that if an assumed size dummy array has n dimensions, the product of the sizes of the first $n-1$ dimensions must be less than or equal to the size of the array, as determined by the above rules.

14.3.4 Use of Functions and Subroutines as Arguments

If a dummy argument is used as a subroutine or function name, the corresponding actual argument must be the name of a subroutine

subprogram if the dummy argument appears in a **CALL** statement, or the name of an intrinsic or external function if the dummy argument appears in a function reference.

Any subroutine or external function name used as an actual argument in a **CALL** statement or function reference must be given in an **EXTERNAL** statement in the program unit in which the name is used as an actual argument.

The **EXTERNAL** statement has the form:

EXTERNAL $name_1, name_2, \ldots, name_n$

where each $name_i$ is the name of a subroutine or an external function that is used as an actual argument in the program unit containing the **EXTERNAL** statement.

For example, if the subroutine

```
SUBROUTINE GREEN(FUN,X,Y,Z)
X=FUN(Y/Z)
RETURN
END
```

is to be called, giving an external function name as actual argument to replace the dummy argument **FUN**, the calling program unit could contain the following statements:

```
EXTERNAL FUN1,FUN2
⋮
CALL GREEN(FUN2,A,B,C)
⋮
CALL GREEN(FUN1,A1,F1,EXP(C))
```

14.3.5 INTRINSIC Statement

An **INTRINSIC** statement is used to identify a name as representing an intrinsic function (see section 14.1.2.1). It also permits a name

representing a specific intrinsic function to be used as an actual argument. The **INTRINSIC** statement has the form:

INTRINSIC $name_1, name_2, \ldots, name_n$

where each *name* is an intrinsic function name.

The use of a name in an **INTRINSIC** statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an **INTRINSIC** statement in that program unit.

The names of the following intrinsic functions must not be used as actual arguments:

- Those for type conversion:

```
INT    IFIX   IDINT  FLOAT  SNGL
REAL   DBLE   CMPLX  ICHAR  CHAR
```

- Those for lexical relationship:

```
LGE    LGT    LLE    LLT
```

- Those for choosing the largest or smallest value:

```
MAX    MAX0   AMAX1  DMAX1  AMAX0  MAX1
MIN    MIN0   AMIN1  DMIN1  AMIN0  MIN1
```

The use of a generic name in an **INTRINSIC** statement does not cause that name to lose its generic property.

A name must not be used in more than one **INTRINSIC** statement in a program unit. Also a name must not be used in both an **EXTERNAL** and an **INTRINSIC** statement in a program unit.

14.4 Transfer of Values between Program Units

Values can be transferred between program units by use of:

- Function values (see section 14.1.2.2).
- Common block items.
- Dummy and actual arguments.

If either of the latter two methods is used for returning results from external functions, care must be used if the actual arguments also appear elsewhere in the calling statement. For example in the statement

```
VAL = A**2/FUN(A,B)*B**2
```

the function **FUN** must not alter the values of its arguments, otherwise the results obtained will be unpredictable.

The second and third methods of transferring values are described in the sections below.

14.4.1 Common block items

A common block is an area of storage that may be referred to in any program unit which mentions the name of the block in a **COMMON** statement. This facility is discusssed fully in section 10.2.3. If in one program unit a value is assigned to an item which forms part of a common block and control is then transferred to another program unit which also refers to that common block, the value assigned in the first program unit becomes the value of the item occupying the same area of storage in the second program.

Items in named common blocks may be given initial values in block data subprograms by means of the **DATA** statement. This is described in section 10.3.2.

14.4.2 Dummy and Actual Arguments

When a transfer of control is made to a subroutine or a function subprogram, actual arguments are supplied in the **CALL** statement

or in the function reference, and the actual arguments are substituted for the dummy arguments in the function or subroutine on entry to the subprogram.

The actual arguments given in the **CALL** statement or the function reference must agree in number, order and type with the dummy arguments that they replace in the subroutine or function subprogram. The correspondence rules for dummy and actual arguments are described more fully in section 14.3.

14.5 Multiple Entry into a Subprogram

It is possible to enter a function or a subroutine subprogram at a statement other than the first executable statement. This is done by using a **CALL** statement or function reference that references an **ENTRY** statement within the subprogram. Control will enter the subprogram at the first executable statement following the **ENTRY** statement.

14.5.1 The ENTRY Statement

The **ENTRY** statement has the form:

ENTRY *name* $(x_1, x_2, \ldots, x_n)$

where:

name is the name of the entry point.

$x_1, x_2, \ldots, x_n$
is a list of dummy arguments corresponding to the actual arguments given in the **CALL** statement or function reference.

The names used as dummy arguments in the list may be used elsewhere in the same subprogram as variables of the same type.

The **ENTRY** statement is non-executable and does not affect control sequencing during the execution of the subprogram.

The appearance of an **ENTRY** statement does not affect the rule that statement functions in a subprogram must precede the first executable statement of that subprogram.

14.5.2 Referencing an ENTRY Statement

The **ENTRY** statement is referenced by a **CALL** statement if it is in a subroutine subprogram or by a function reference if it is in a function subprogram.

A subprogram must not reference itself directly or through any of its own entry points.

The actual arguments in the **CALL** statement or function reference must agree in order, number and type with the dummy arguments in the **ENTRY** statement being referenced. However, the dummy arguments of the **ENTRY** statement need not agree in order, type or number with the dummy arguments in the **SUBROUTINE** or **FUNCTION** statement or in any other **ENTRY** statement in the subprogram.

In a function subprogram the types of the function name and entry point name are determined by the **FUNCTION** and **ENTRY** statements. If an entry name in a function subprogram is of type character then each entry name and the name of the function subprogram must be of type character. If not of type character the types of the function name and entry point names can be different; whether they are or not, they are treated as variables equivalenced by means of an **EQUIVALENCE** statement (see section 10.2.4). After one of these variables is assigned a value in the subprogram, the others become undefined.

If information for an array is passed in the reference to an **ENTRY** statement, the array name and all its dimension parameters (except any that are in a common area or are constant) must appear in the

dummy argument list of the **ENTRY** statement. A name that appears as a dummy argument in an **ENTRY** statement must not appear in any executable statement preceding that **ENTRY** statement.

14.5.3 Entering the Subprogram

Entry into a subprogram assigns new values to the dummy arguments of the referenced **ENTRY** statement. Thus, all appearances of these arguments in the whole subprogram are affected.

Reference to an **ENTRY** statement will not transmit new values for the arguments not listed in that **ENTRY** statement.

Entry cannot be made into an **IF**-block or the range of a **DO** statement.

14.5.4 Exit from the Subprogram

On exit from the subprogram, the value returned to the calling program is the last value assigned in the subprogram to the entry point name before control is returned. A value may be returned via a name other than the one used to enter the subprogram. If this is done the two names must be of the same type, otherwise the value returned will be undefined.

14.6 The SAVE Statement

In Parallel Fortran, variables do not retain their values after a subroutine or function returns to its calling subprogram. If the subroutine or function is entered again, the values of variables may have changed. There are the following exceptions to this:

- All variables in **COMMON** blocks;

- Arrays;
- Variables which have been intialised by means of a **DATA** statement or an extended explicit type specification statement.

The **SAVE** statement can be used to make items retain their values after the execution of a **RETURN** or **END** statement. It has this format:

SAVE $name_1, name_2, \ldots, name_n$

where each $name_i$ is a named common block name preceded and followed by an oblique; a variable name; or an array name.

A name may not occur more than once in a **SAVE** statement within a particular program unit. Dummy argument names, procedure names and names of items in a common block must not appear in a **SAVE** statement.

A **SAVE** statement without a list is treated as though it contained the names of all allowable items in that program unit.

The appearance of a common block name preceded and followed by an oblique in a **SAVE** statement has the effect of specifying all of the items in that common block. If a particular common block name is specified by a **SAVE** statement in a subprogram it must also be specified by a **SAVE** statement in every other subprogram in which that common block appears. Note however, that if a named common block is used in the main program unit it is effectively saved for all subprograms referenced by the main program.

The ANSI Fortran 77 standard specifies that only the following must retain their values between calls to a subprogram:

- Items specified by **SAVE** statements.
- Items in blank common.
- Items defined with an initial value and not assigned a new value.

- Items in a named common block that appears in the subprogram and in at least one other program unit which is referencing, either directly or indirectly, that subprogram.

Although Parallel Fortran by default saves more than this minimum, programs which are intended to be portable should make use of **SAVE** statements, so as to avoid problems with implementations which do not.

14.7 The INCLUDE Statement

In Parallel Fortran the compiler may be directed to include the contents of secondary source files in the compilation. The effect of using the file inclusion mechanism is identical to having the text of the secondary file or files present in the file being compiled.

The directive to include a file is a special Fortran statement:

```
INCLUDE 'filename'
```

Note that because the filename is a Fortran character constant, the character '\' is interpreted as an escape character, as described in section 9.2.1.11. This means that if you wish to use an MS-DOS pathname when specifying the included file, you must double the '\' character. For example:

```
INCLUDE 'c:\\lib\\incfiles\\maths.inc'
```

A description of where the compiler looks for the specified file may be found in section 17.3.

Not
For
Sale

Chapter 15

Format Specification

In Fortran 77 all input and output data are handled in the form of records.

Records can be output to:

- the display;
- a printer;
- files on magnetic media;
- internal files consisting of character variables or arrays.

Records can be input from:

- the keyboard;
- files on magnetic media;
- internal files

Two kinds of records are recognised: formatted records and unformatted records. A *formatted record* is a sequence of characters which may, for instance, be input from the keyboard or output as a line of print. Each record can be regarded as being split into

fields, where each field contains one or more characters and normally represents the value of one variable or array element.

An *unformatted record* normally contains one or more values which are in internal machine form and is commonly used for re-input into the computer rather than for visual inspection of program results.

The reading or writing of a formatted record may be controlled by a *format specification*. The format specification can be given either in a **FORMAT** statement (see section 15.2.1) or as values of character arrays, character variables, or other character expressions.

A format specification defines the form of one or more records in either an external file or an internal file and specifies the transformation to be applied between the internal machine form and the fields of a record. When reading a formatted record the format specification describes the fields within the record and the manner in which these are to be converted from character codes on the input medium into internal machine form. If the format specification is used in association with an output statement it describes the manner in which data in internal machine form is converted to records in the character code used by the output medium.

The individual fields within a record and the conversions to be applied to them are specified within a format specification by means of *edit descriptors*, which take the forms described in section 15.3. Thus, for example, the edit descriptor **I4** describes an integer field four character positions wide. The way in which edit descriptors can be combined to form complete format specifications is described below, and the use of edit descriptors in **FORMAT** statements and arrays is discussed in sections 15.2.1 and 15.3.

Input normally involves assigning the value represented in each field to a variable or array element in store, output normally involves placing the value of a variable or array element in store into the appropriate field. Some descriptors have other effects such as allowing for spacing between fields. The edit descriptors of a format

specification are associated with items in an input or output list in a `READ` or `WRITE` statement as described in section 15.2.3.

The ANSI Standard describes the first character of a formatted output record as a print control character which controls the vertical spacing of the record if it is sent to a display device such as a printer or screen; this is discussed in section 15.3.1.10.

15.1 Format Specifications

A format specification consists of a series of edit descriptors (see section 15.3) surrounded by parentheses. The specification describes one or more records which are to be input or output. Apart from the edit descriptors, the following may also appear in the specification:

- commas
- parentheses
- repeat counts and group repeat counts

15.1.1 Field Separators

Consecutive edit descriptors and groups of edit descriptors enclosed in parentheses (see section 15.1.3) must be separated by a comma, except

- between an edit descriptor containing a `P` and an immediately following edit descriptor containing an `F`, `E`, `D` or `G` descriptor.
- before or after a slash.
- before or after a colon.

Other slashes may appear before or after a series of edit descriptors. A comma marks the end of a field. A slash marks the end of a field and the end of a record (see below).

15.1.2 Slash Editing

A slash '/' marks the end of a field and the end of a record. Thus, when one slash is present the current record is terminated and processing of the next record begins: on input any remaining data in the current record will be ignored while on output no more data will be written to the record.

If two or more consecutive slash edit descriptors appear in a format specification then the first slash will terminate the current record and each succeeding slash will on input, skip a record, or on ouput write an empty record. For example, the format specification

```
(F1,F2/F3,///F4)
```

where each F is an edit descriptor, describes a series of records in the following order:

- a record containing two fields corresponding to F1 and F2.
- a record containing one field corresponding to F3.
- two blank records (or two records to be skipped on input).
- a record containing one field corresponding to F4.

The parentheses at the beginning and end of the format specification may be considered to initiate a new record and terminate a record respectively. For example, the specification

```
(///F///)
```

where F is an edit descriptor, describes a series of records in the following order:

- three blank records.
- a record containing one field corresponding to F.
- three blank records.

15.1.3 Repetition of Descriptors

An edit descriptor or a group of edit descriptors (that is a series of descriptors enclosed in parentheses) may be repeated by preceding it with an integer *r*. The effect will be as if the descriptor or group was repeated *r* times. Non-repeatable edit descriptors (see section 15.3) may be repeated only if they are enclosed in parentheses.

The repeat count or group repeat count *r* must be a positive (and non-zero) integer. If a group enclosed in parentheses is not preceded by a group repeat count, a count of one is assumed.

Any group of descriptors enclosed in parentheses may have among its items other groups of descriptors enclosed in parentheses but these must not be nested more than seven deep.

For example, the specification

```
(F1,3F2/5F3,6(/F4,F5),2F6)
```

where each F is an edit descriptor, describes a series of records in the following order:

- a record containing one field described by F1 and three described by F2.
- a record containing five fields described by F3.
- five records each containing one field described by F4 and one by F5.
- a record containing one field described by F4, one by F5 and two by F6.

15.2 Format Specification Methods

Format specifications as described in the earlier part of this chapter may be either specified in FORMAT statements or as values of character

arrays, character variables, or other character expressions. When a format specification is to be used to control the input or output of data, either a reference is made to the label of the **FORMAT** statement or the name of the character array or character variable, or a character expression is used.

Section 15.2.2 describes character format specifications, section 15.4 contains examples, and information on the use of format specifications for input and output is given in chapter 16.

15.2.1 The FORMAT Statement

A **FORMAT** statement is a non-executable statement and has the form:

label **FORMAT** *specification*

where:

label is a statement label; every **FORMAT** statement must be labelled.

specification is a format specification as described earlier in the chapter.

The following are examples of **FORMAT** statements, where each **F** is an edit descriptor as described in section 15.3. Complete examples of **FORMAT** statements including the use of edit descriptors are given in section 15.4.

```
16    FORMAT (F1)
3049  FORMAT (F2,3F3/F4)
4     FORMAT (F5,F6,6(/F7,3F8)/F9)
```

15.2.2 Character Format Specification

A format specification may be held in a character array, or within a character variable, or may be specified as a character expression. The

specification must include the parentheses at the beginning and end. It may have been read into a character array or character variable by means of an **A** conversion code (see section 15.3.1.7) or may have been initialised by a **DATA** statement, or assigned during the course of the execution of the program. Character data may follow the right parenthesis that ends the format specification and will be ignored.

For example, the format specifications given in the examples in the previous section could be held in arrays instead of being given in **FORMAT** statements. If the specification given in the statement labelled **3049** were held in a **CHARACTER*4** array named **ARR**, the first element of the array would hold the left parenthesis followed by some or all of the characters that make up the edit descriptor **F2**. The succeeding character positions would hold the comma, the figure three, the characters making up the field descriptor **F3**, the slash, the figure three, the characters making up the edit descriptor **F4** and, finally, the right parenthesis, using as much of the remainder of the array **ARR** as is required.

A character format specification can be used for input or output in the same way as one given in a **FORMAT** statement. For example, a reference to the array **ARR** mentioned in the previous paragraph would have the same effect as a reference to the statement labelled **3049** in the previous section.

Formats may be varied at run-time either by assigning new values to, or using **A** format codes (see section 15.3.1.7) to read values into character array elements or character variables.

For compatibility with ANSI66, Parallel Fortran also allows arrays which are not of type character to define a format specification.

15.2.3 Effect of FORMAT Statements and Character Format Specifications

A **READ** or **WRITE** statement (see sections 16.3.1 and 16.4.1) referencing a **FORMAT** statement or character format specification normally

contains a list of variable names and array elements known as an input or output list. These are associated in order with the edit descriptors in the format specification, except that non-repeatable edit descriptors are not associated with any variable or array element. Thus if a list of names in a **READ** statement was

```
Y,Z
```

and the **FORMAT** specification was

```
(F1,W,F2)
```

where **F1** and **F2** are repeatable edit descriptors and **W** is a non-repeatable edit descriptor, then the variable **Y** will be associated with edit descriptor **F1** and **Z** with edit descriptor **F2**.

Each action performed during the execution of a formatted **READ** or **WRITE** statement is determined by the next descriptor in the format specification and the next item, if any, in the input or output list. If the descriptor is a non-repeatable edit descriptor, then it is acted upon and the next descriptor is examined; this process is repeated until a repeatable edit descriptor is encountered.

The descriptor must be one which is permitted with a variable or array element of the type under examination (see section 15.3 below). On input the value represented in the field is converted according to the edit descriptor and is assigned to the variable or array element; on output the value of the variable or array element is output to the field in the format specified. The next item from the list in the **READ** or **WRITE** statement and the next edit descriptor are then selected, and the process is repeated until the input or output list is satisfied.

When the last named variable or array element has been operated upon, the next descriptor is examined. If it is a non-repeatable edit descriptor it will be acted upon and the next descriptor will be examined in the same way. This process is repeated until a repeatable edit descriptor is encountered.

When the last edit descriptor has been acted upon or when an edit descriptor not of the types given above is encountered, execution of

the statement ceases.

A special case of a formatted **READ** or **WRITE** statement is one that does not contain a list of variable and array element names; the first or only descriptor in the corresponding format specification should be a non-repeatable edit descriptor otherwise the corresponding record is skipped.

If, when the format specification has been completely scanned, there are still items left in the list of names, a new record will be started and the format specification will be re-scanned as follows:

- If there are no internal parentheses, scanning will be repeated from the beginning of the specification;
- If there are internal parentheses, scanning will be repeated from the left parenthesis corresponding to the right-most internal right parenthesis. If this left parenthesis is preceded by a group repeat count, the repeat count is taken into account.

This process is repeated whenever the closing parenthesis of the format specification is rescanned and items still remain in the input or ouput list.

The symbol ⇑ in the following examples shows where scanning would be restarted:

```
FORMAT(.........)
       ⇑

FORMAT(...,(...))
           ⇑

FORMAT(...,(...(...)...)...)
            ⇑

FORMAT(...(...)...3(.(.).(.))..)
                    ⇑
```

Examples of complete **FORMAT** statements and their effects are given in section 15.4.

15.3 Edit Descriptors

The edit descriptors are used to specify the external format of fields in a record, and are classified into two types:

- repeatable edit descriptors.
- non-repeatable edit descriptors.

Repeatable edit descriptors may be preceded by a repeat count which is an unsigned, non-zero, integer constant and which specifies the number of times the edit descriptor is to be repeated. If the repeat count is omitted a value of one is assumed. A repeatable edit descriptor indicates the manner in which a variable or array element is to be edited, and contains one of the following format codes: **A**, **D**, **E**, **F**, **G**, **I**, **L** and (in Parallel Fortran), **Z** and **O**.

Repeatable edit descriptors and their effects are described in sections 15.3.1 to 15.3.1.9. They generally operate on fields whose width is defined by the edit descriptor. However, in Parallel Fortran, a comma in data being read may be used to override the defined field width. Thus if a field contains a comma, the comma delimits that field and any remaining data in the field will be associated with the following field. This facility may be used with **D**, **E**, **F**, **G**, **I**, **L**, **O**, and **Z** edit descriptors; a comma in a field read by an **A** edit descriptor has no special effect.

For example, if the formal specification

```
(L6, I3, E15.8, A4)
```

were used to read the following record

```
.T,-4,␣␣␣␣␣252,A,B,C
```

then the following values would be assigned to the corresponding items in the input/output list:

```
.TRUE.
-4
```

```
0.252E-05
A,B,
```

From this example one should note that the use of the comma within a field has no effect on either the data read by the **A** conversion code, or on the decimal point positioning as defined by the **E** conversion code.

It is not possible to extend the width of a field using this facility.

Non-repeatable edit descriptors are described in sections 15.3.1.10 to 15.3.1.16. Non-repeatable edit descriptors, slashes and colons must not be preceded by a repeat specification, and they operate independently of any items in the input or output list. A non-repeatable edit descriptor contains one of the following format codes: **H**, *'literal'*, **T**, **TL**, **TR**, **X**, **S**, **SP**, **SS**, **P**, **BN**, **BZ** and (in Parallel Fortran), **$** or **Q**.

Non-repeatable edit descriptors are generally used to control the manner in which a field is to be edited, although they may be used for other purposes such as altering the positioning within a record at which transfer of data is to start. The use of a colon to terminate format control if there are no more items in the input or ouput list is described in section 15.3.2.

The different edit descriptors, the types of internal variables with which they correspond, and their actions, are listed in table 15.1. A reference is given to the section in which each descriptor is discussed.

In the list of edit descriptors shown in table 15.1, the capital letters **I**, **F**, **E**, **D**, etc., are called format codes and it is these format codes that indicate in what way the record is to be converted either from external format to internal machine form on input, or from internal machine form to external format on output. The other symbols should be interpreted as follows:

w specifies the number of character positions occupied by a field in the record.

Edit descriptor	Internal data type	Action	Section
I*w* **I***w*.*m*	Integer, byte	Numeric conversion	15.3.1.1
F*w*.*d*	Real, double precision, part of complex or double complex	Numeric conversion	15.3.1.2
D*w*.*d*			15.3.1.3
E*w*.*d*			15.3.1.3
E*w*.*d***E***e*			15.3.1.3
G*w*.*d*			15.3.1.4
G*w*.*d***E***e*			15.3.1.4
*k***P**		Scaling real numbers	15.3.1.5
L*w*	Logical	Logical value conversion	15.3.1.6
A **A***w*	Character	Character conversion	15.3.1.7
Z*w* **Z***w*.*m*	Any	Hexadecimal conversion	15.3.1.8
O*w* **O***w*.*m*	Any	Octal conversion	15.3.1.9
*n***H** '*literal*'		Write text	15.3.1.10
*n***X**		Alter position in	15.3.1.11
T*c*		record where	15.3.1.12
TL*c*		transfer of data	15.3.1.12
TR*c*		should begin	15.3.1.12
S		Control of optional	15.3.1.13
SP		plus characters in	15.3.1.13
SS		numeric output fields	15.3.1.13
BN **BZ**		Control handling of input blanks	15.3.1.14
Q	Integer or Logical	Query size of rest of record	15.3.1.16
$		Input prompt	15.3.1.15

Table 15.1: Edit Descriptors

d specifies the number of character positions in the fractional part of a number.

m specifies the number of significant digits in the field.

e specifies the number of digits in the exponent.

k specifies an optionally signed integer constant.

w, *e*, *n* and *c* are nonzero, unsigned, integer constants.

Since complex values can be considered as two real values for the purposes of input and output, they are transferred by means of two D, E, F, or G format codes; one format code for the real part and one for the imaginary part. Double complex values can similarly be considered as two double precision values, and they can be transferred in the same way.

15.3.1 Format (Conversion) Codes

The following sections describe the various format codes and their effects on data on input and output. The term conversion code is used interchangeably with format code for those codes which are directly concerned with converting data between its external format and its internal machine form.

15.3.1.1 The I Conversion Code

The I conversion code is used to transfer integer data. The edit descriptor has one of the following forms:

I*w*
I*w*.*m*

where:

w is an unsigned positive integer that defines the width of the field in characters.

m is an unsigned integer that gives the minimum number of digits to be output.

Input

The I*w*.*m* edit descriptor is treated identically to the I*w* edit descriptor. When this conversion code is used for input, the next *w* characters in the current record are read as an integer and the value is assigned to the next variable or array element in the input list.

The characters in the external field may be a signed or an unsigned integer; unsigned numbers will be assumed to be positive. Spaces before the first digit are ignored but must be included in the character count *w*, as must the sign, if any. All other spaces are treated as zeros or are ignored as determined by a combination of any **BLANK=** specifier that is currently in effect for the unit (see chapter 16), and any **BN** or **BZ** edit descriptors (see section 15.3.1.14). Unless specified otherwise spaces, other than leading spaces, are treated in the examples which follow as zeros. A field of all spaces is treated as a field of zeros. The field must not contain a decimal point or an exponent.

Note that if the next element in the input list is a byte variable or array element, the value read is checked before it is assigned, and if it is outside the range −128 to +127 an error will be reported.

The following examples demonstrate the effects of the I format code on input. The symbol '␣' represents a space.

Edit descriptor	External number	Internal number
I5	␣+376	+376
I6.2	␣␣␣␣-2	−2
I6	␣␣␣␣-2	−2
I4	34␣␣	+3400
I3	␣␣␣	0

Output

When this conversion is used for output, the edit descriptor will cause the value of the corresponding item in the output list to be output as an integer occupying w character positions in the current record; the number will be right justified. The effect of using the Iw.m edit descriptor is the same as Iw except that the unsigned integer constant consists of at least m digits and if necessary, has leading zeros.

The value of m must not exceed the value of w. If m is zero and the value of the output item is zero then the output field consists only of blank characters regardless of the sign control that is currently in effect. Negative numbers will be preceded by a minus sign which occupies one of the w character positions specified; positive numbers will be unsigned. The field of w characters will be space filled on the left if necessary. If the integer to be output, including any minus sign, exceeds w characters, the output field is filled with asterisks.

Some examples follow of the effect of the I code on output:

Edit descriptor	Internal number	External number
I5	+3659	␣3659
I6	−987	␣␣-987
I3	+3659	***
I6.4	−123	␣-0123
I5.3	+24	␣␣024

For example, the following:

```
      I = 20
      J = -236
      K = 9872
      WRITE(6,100) I,J,K
100   FORMAT('0',I4.3,2I8)
      ⋮
```

causes the following line to be printed:

```
␣020␣␣␣␣-236␣␣␣␣9872
```

15.3.1.2 The F Conversion Code

The **F** conversion code is used to transfer real numbers without an exponent. It may also be used to input real numbers with an exponent. The edit descriptor has the form:

F*w*.*d*

where:

w is an integer giving the width in characters of the external field.

d is the number of digits in the fractional part of the number. *w* must always be greater than or equal to *d*.

Input

When this conversion code is used for input, the edit descriptor causes the next *w* characters in the current record to be read as a real number and the converted value to be assigned to the next item in the input list. If the item is of type complex or double complex then two edit descriptors are required.

The external field should contain *w* characters and include :

- a sign (optional).
- a string of digits which may contain a decimal point.

The external field may also contain an exponent.

Unsigned numbers are assumed to be positive. Spaces occurring before the first digit are ignored. All other spaces are treated as for I editing. All spaces must be included in the character count. A field of all spaces is treated as zero.

If the field does not contain a decimal point, the number is treated as though a point occurred before the last *d* digits of the string. This is the number that any scale factor can be considered to operate on (see section 15.3.1.5). If the external field contains a decimal point, this will override the decimal point implied by the value *d* in the descriptor.

Some examples follow of the effects of the F format code on input. The symbol '␣' represents a space.

Edit descriptor	External number	Internal number
F6.2	␣␣1234	12.34
F6.2	1.2300	1.23
F6.2	␣-2345	−23.45
F6.2	␣123E1	12.30

Output

When this conversion code is used for output, the edit descriptor causes the value of the next item in the output list to be output as a decimal fraction, rounded to *d* decimal places and made up by trailing zeros if necessary. The number is right justified and if it is negative it is preceded by a minus sign. The field will be space-filled on the left to make up the *w* characters. If the number has no integral part and if the field width specified is large enough, the decimal point will be preceded by a zero. The decimal point and minus sign must be included in the character count *w*. If the item

is of type complex or double complex, two descriptors are required to output it.

The number of characters to be output should not exceed the field width. If it does, the output field is filled with asterisks.

Similarly, in Parallel Fortran, if an attempt is made to output the exceptional IEEE values NaN, $+\infty$ or $-\infty$, the output field is filled with question marks: see section 11.1.10.

Some examples follow of the effects of the **F** code on output. The symbol '␣' represents a space.

Edit descriptor	Internal number	External number
F10.4	+5227.3278	␣5227.3278
F10.4	−345.6789	␣-345.6789
F10.4	+12.3	␣␣␣12.3000
F10.4	−3.21989623	␣␣␣-3.2199

For example, the following statements:

```
      X = -3.7690
      Y = 1.55
      Z = 12345.69
      WRITE(6,100) X,Y,Z
100   FORMAT(1H␣,F10.4,F8.6,F8.3)
      ⋮
```

would produce the following line:

```
␣␣␣-3.76901.550000********
```

The value for Z requires five positions before the decimal point but since only four are available the value is represented by ********, that is asterisks in all eight positions of the field.

15.3.1.3 The E and D Conversion Codes

The E and D conversion codes are used to transfer real numbers. The edit descriptors have the following forms:

E*w*.*d*
D*w*.*d*
E*w*.*d***E***e*

where:

w	is an integer giving the width of the field in characters.
d	is an integer giving the number of digits in the fractional part of the number. *w* must always be greater than or equal to *d*.
e	is an integer giving the number of digits in the exponent: *e* must be greater than zero. It has no effect on input.

Input

When the E or D conversion code is used for input, the edit descriptor causes the next *w* characters in the current record to be read as a real number and the converted value to be assigned to the next variable or array element in the input list. If the item is of type complex or double complex then two edit descriptors are required.

The external field should contain *w* characters and include:

- a sign (optional).
- a string of digits which may contain a decimal point;
- an exponent (optional).

The exponent may have one of the following forms;

- a signed integer constant.

- E or D followed by a signed integer constant;
- E or D followed by an unsigned integer constant.

Unsigned numbers and exponents are assumed to be positive. Spaces occurring before the first digit are ignored. All other spaces are treated as for I editing. All spaces must be included in the character count. A field of all spaces is treated as zero.

If the field does not contain a decimal point, the number is treated as though a point occurred before the last *d* digits of the string. This is the number that any exponent or scale factor can be considered to operate on. If the external field contains a decimal point, this will override the decimal point implied by the value d in the descriptor.

Some examples follow of the effects of the E format code on input. The symbol '␣' represents a space.

Edit descriptor	External number	Internal number
E7.3	7654321	+7654.321
E7.3	␣+137-3	+.000137
E7.3E2	␣+137-3	+.000137
E7.3	1.234E2	+123.4
E7.3	-123E02	−12.3

Output

When the E or D conversion code is used for output, the edit descriptor outputs the next item in the output list as a decimal fraction with an exponent. If the exponent field *e* exists then the exponent will be output with *e* digits.

The fractional part, f, of the external number will be in the range $0.1 \leq f < 1$ and will be rounded to *d* digits and will be output preceded by a minus sign (if the number is negative), a zero (if the field width specified is wide enough) and a decimal point. If the item is of type complex or double complex then two edit descriptors are required.

The exponent will have one of the forms:

E$\pm d_1 d_2$	if E$w.d$ or D$w.d$ is used and $\|exp\| \leq 99$
$\pm d_1 d_2 d_3$	if E$w.d$ or D$w.d$ is used and $99 < \|exp\| \leq 999$
E$\pm d_1 d_2 \ldots d_n$	if E$w.d$Ee is used

where $d_1\ d_2 \ldots d_n$ are digits. The number will be right justified.

The fractional part f will be signed only if it is negative. The exponent part will always be signed. The scale factor can be used to alter the range of the fractional part f of the external number from the limits defined above.

If necessary, the field will be space filled on the left to w characters. The number of characters, including the minus sign if any, should not exceed the field width. If it does, the output field is filled with asterisks.

Similarly, in Parallel Fortran, if an attempt is made to output the exceptional IEEE values NaN, $+\infty$ or $-\infty$, the output field is filled with question marks: see section 11.1.10.

Some examples follow of the effects of the E code on output. The symbol '␣' represents a space.

Edit descriptor	Internal number	External number
E14.5	+12345678	␣␣␣0.12346E+08
E14.5	−1.23	␣␣-0.12300E+01
E14.5	+.000123	␣␣␣0.12300E-03
E14.5	−.003	␣␣-0.30000E-02
E14.5E4	−.003	-0.30000E-0002

Examples

The following:

```
A = 4764.732
B = -21.5E-4
C = .003210
```

```
      D = -99.9E3
      WRITE(6,100) A,B,C,D
100   FORMAT('0',E15.8E3,E13.6,E12.4,E9.4)
      ⋮
```

causes this line to be printed out:

```
0.47647320E+004-0.215000E-02␣␣0.3210E-02*********
```

The value for D requires at least ten positions ($-.9990 \times 10^5$) and as only nine are specified, the field is set to '*********'.

The following:

```
      DOUBLE PRECISION X,Y,Z
      X = -3.66D2
      Y = 123456.12345
      Z = 155.151
      WRITE(6,100) X,Y,Z
100   FORMAT('0',D10.3,D16.8,D18.7)
      ⋮
```

causes this line to be printed out:

```
-0.366E+03␣␣0.12345612E+06␣␣␣␣␣0.1551510E+03
```

The edit descriptor for Y specifies only eight significant figures; in this case rounding occurs.

The following:

```
      REAL X, Y
      X = -1.0
      Y = SQRT (X)
      WRITE (6,100) Y
100   FORMAT(F5.2)
      ⋮
```

causes the following to be output:

```
?????
```

This is because the result of performing a SQRT of -1.0 is a NaN.

15.3.1.4 The G Conversion Code

The G conversion code is used to transfer a real or double precision value, or the real or imaginary part of a complex or double complex value. Edit descriptors using the G conversion code have the format:

```
Gw.d
Gw.dEe
```

where:

w	is an integer giving the width, in characters, of the external field.
d	is an integer giving the number of digits in the fractional part of the number: *w* must always be greater than or equal to *d*.
e	is an integer giving the number of digits in the exponent. It has no effect on input.

Input

For input the G conversion code has the same effect as if it were E*w*.*d*, D*w*.*d* or F*w*.*d* (see sections 15.3.1.2 and 15.3.1.3).

Output

When this conversion code is used for output the edit descriptor outputs the next item in the output list either in fixed point form (without an exponent) or in floating point form (with an exponent). The magnitude of the value determines the form in which it is output as follows:

- If the number, x, is outside the range $0.1 \leq x < 10^d$, then the number is output with an exponent in the same manner as the E edit descriptor (see section 15.3.1.3).

- If the number is inside the above range, then the d most significant digits of the number are output as a decimal fraction without a decimal exponent and will be justified towards the left by a fixed number of spaces. If the Gw.dEe conversion code is used then $e+2$ spaces will be produced at the right of the field; four spaces will be produced if the Gw.d conversion code is used. The field width w must allow for these additional characters.

- If a scale factor (see section 15.3.1.5) is operating, it will have no effect unless the value being output is outside the range $0.1 \leq x < 10^d$. If the value is outside this range, then the effect of the scale factor will be as for the E conversion code (see section 15.3.1.3).

Some examples follow of the effects of the G code on output. The symbol '␣' represents a space.

Edit descriptor	Internal number	External number
G11.4	+10.3456	␣␣10.35␣␣␣␣
G11.4E4	+10.3456	10.35␣␣␣␣␣␣
G11.4	−0.000367	-0.3670E-03
G11.4E1	−0.000367	␣-0.3670E-3
G11.4	+4958.67	␣␣4958.␣␣␣␣
G11.4	+49586.7	␣0.4959E+05
2PG11.4	+10.3456	␣␣10.35␣␣␣␣
2PG11.4	−.00036	␣-36.00E-05

Example

The following:

```
      REAL A,R,S,T
      COMPLEX C
      READ(5,4) A,C,R,S,T
4     FORMAT(2G8.3,G6.2,G11.8E2,G4.0,G15.12)
      WRITE(6,41) A,C,R,S,T
41    FORMAT(1H␣,G10.3,G9.2,2G12.5E3,G13.6,G7.1)
```

and a data input record of the form:

```
␣33854␣␣2000.E-4-12775␣␣␣-96612E-8768␣+␣␣␣105␣␣␣␣␣␣␣␣
```

would produce a printed output line as follows:

```
␣0.339E+04␣0.20␣␣␣␣␣-127.75␣␣␣␣␣␣-.96612E-011␣␣7680.00␣␣␣␣␣0.1E-01
```

15.3.1.5 The Scale Factor

The scale factor is used to change the position of the decimal point in real numbers. It has the form:

```
kP
```

where *k* is an integer, optionally preceded by a minus sign.

A scale factor of zero is assumed in any format specification until a scale factor is specified. Once a scale factor is specified it operates on all real or complex values converted in that **FORMAT** statement by **F**, **E**, **D** or **G** edit descriptors (see sections 15.3.1.1, 15.3.1.2 and 15.3.1.4) until a new scale factor is encountered. A scale factor of the form

```
0P
```

cancels the operation of any previous scale factor.

Effects on Input

A scale factor affects only real numbers without an exponent. The scale factor is ignored for any other type of number.

The effect of the scale factor on a real number input is that the number will be divided by 10^k as it is converted from an external value to the internal value.

That is:

- if the input data is in the form *ab.cde* and it is required to use this data internally in the form *.abcde*, the edit descriptor necessary would be `2PF6.3`

- if the input data is in the form *ab.cde* and it is required to use this data internally in the form *abcd.e*, the edit descriptor would be `-2PF6.3`

Effects on Output

The scale factor can be used to modify the effect of edit descriptors containing `F`, `D`, `E` or `G` conversion codes as follows:

- `F` conversion code:

  The internal number is multiplied by 10^k as it is output.

- `E` or `D` conversion code:

  The internal number is multiplied by 10^k as it is output but the exponent is adjusted to compensate. Thus the number is changed in form but not in value. Note that in this instance the scale factor k must be restricted to the range $-d < k < d+2$, where d is an integer which defines the number of digits in the fractional part of the number.

- `G` conversion code:

  If the number is output without an exponent, the scale factor has no effect. Otherwise, the effect is the same as for the `E` or `D` descriptors.

The scale factor has no effect on edit descriptors other than these.

Examples

The following table shows the effect of a scale factor on field descriptors used for input:

Edit descriptor	External number	Internal number
-3PF6.3	99.99	99990.0
3PF6.3	99.99	.09999
2PF12.2	4120.0	41.2
0PF5.2	21.2	21.2
2PE7.1	8642.0	86.42
2PE7.1	86.42E2	8642.0

The next table shows the effect of a scale factor in format codes used for output:

Edit descriptor	Internal number	External number
2PF11.0	12345.0	␣␣␣1234500.
-3PE11.5	12345.0	0.00012E+08
2PE11.3	12345.0	␣␣12.34E+03
4PG11.3	12345.0	␣␣1234.E+01
-1PG11.3	12345.0	␣␣0.012E+06

15.3.1.6 The L Conversion Code

The L conversion code is used to transfer logical values. The edit descriptor has the form

L*w*

where *w* is an integer giving the width in characters of the external field.

Input

When this conversion code is used for input, the edit descriptor will cause a field of *w* characters to be read and converted to the internal representation of .TRUE. or .FALSE.; the converted value is assigned to the corresponding item in the input list. The external field consists of w characters as follows:

- optional spaces, optionally followed by a decimal point, followed by T (representing the value .TRUE.), optionally followed by any other characters, or
- optional spaces, optionally followed by a decimal point, followed by F (representing the value .FALSE.), optionally followed by any other characters.

Output

When this conversion code is used for output it will cause $w - 1$ spaces to be output followed by the character T if the next item in the output list item has the value .TRUE. or by the character F if the item has the value .FALSE. For example, the following:

```
      LOGICAL X,Y
      X = .TRUE.
      Y = .FALSE.
      N = 250
      A = 27.4
      WRITE(6,4) N,X,A,Y
4     FORMAT('0',I5,L6,F6.2,L3)
      ⋮
```

will cause the following line to be output:

```
␣␣250␣␣␣␣␣T␣27.40␣␣F
```

15.3.1.7 The A Conversion Code

The A conversion code is used to transfer character values. The edit descriptor has one of the following forms:

```
Aw
A
```

where w gives the width, in characters, of the external field. If no width is specified then the number of characters in the external field is the length of the item in the input/output list.

Although the ANSI Standard allows the **A** format code to be used only with data of type character, Parallel Fortran allows the format code to be used with any data type.

Input

If the field width *w* is greater than or equal to the length of the item (*len*), then the rightmost *len* characters will be taken from the input field. If the field width is less than *len* then *w* characters will be left justified in the field with *len* – *w* padding spaces to the right. For example, the following:

```
      CHARACTER*4 A,B,C,X,Y,Z
      READ(5,3) A,B,C,X,Y,Z
3     FORMAT(A4,A,A,A5,A2,A5)
```

and this data input record:

```
HOT*AND+COLDHOT*AND+COLD
```

would cause the character strings to be assigned to the variables, **A**, **B**, **C**, **X**, **Y**, **Z** as follows:

Variable	String
A	HOT*
B	AND+
C	COLD
X	OT*A
Y	ND␣␣
Z	COLD

Output

When this conversion code is used for output, the edit descriptor will cause *w* characters to be output. If *w* is less than or equal to *len*, then the *w* leftmost characters are output and the rest are ignored. If *w* is greater than *len* then *w* – *len* blank characters will be output followed by the characters of the list item.

15.3.1.8 The Z Conversion Code

The Z conversion code is used to transfer hexadecimal data. The edit descriptor has one of the following forms:

```
Zw
Zw.m
```

where:

w is an unsigned positive integer which defines the width in characters of the external field;

m is an unsigned integer which gives the minimum number of digits to be output.

The Z conversion code is not allowed by the ANSI Standard. Parallel Fortran allows the format code to be used with any data type.

Input

The Z*w.m* edit descriptor is treated identically to the Z*w* edit descriptor. The edit descriptor will read the next *w* characters in the current record as hexadecimal data and the converted value will be assigned to the relevant item in the input list.

Within an external field all spaces before the first hexadecimal digit will be ignored, but must be included in the character count *w*. All other spaces are treated as for I editing. Some examples follow of the effect of the Z code on input.

Edit descriptor	External number	Internal number
Z3	100	256
Z3	␣␣F	15
Z3	A0A	2570
Z3.2	A0A	2570

Output

When used with an output statement, this edit descriptor will output the next item in the output list as a hexadecimal number occupying *w* character positions in the current record. The Z*w.m* edit descriptor has a similar effect except that the hexadecimal number consists of at least *m* digits and, if necessary, has leading zeros.

The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the output item is zero, then the output field will consist only of space characters. The hexadecimal number will be right justified within the field by inserting spaces on the left if necessary. If the value to be output exceeds *w* characters, the output field is filled with asterisks.

Some examples follow of the effect of the Z code on output.

Edit descriptor	Internal number	External number
Z4	10	␣␣␣A
Z4.3	10	␣00A
Z4.0	0	␣␣␣␣
Z4	65536	****

15.3.1.9 The O Conversion Code

The O conversion code is used to transfer octal data. The edit descriptor has one of the following forms:

O*w*
O*w*.*m*

where:

w is an unsigned positive integer describing the width of the field in characters.

m is an unsigned integer which defines the minimum number of digits to be output.

The ANSI Standard does not allow the O format code; in Parallel Fortran the format code may be used with any data type.

Input

The O*w*.*m* edit descriptor has the same effect as the O*w* edit descriptor. When this conversion code is used for input, the edit descriptor will read the next *w* characters as octal data and will assign the converted value to the next variable or array element in the input list. Spaces before the first octal digit in a field will be ignored but must be included in the character count *w*. All other spaces are treated as for I editing. Examples follow of the effects of the O format code on input; the symbol '␣' represents a space.

Edit descriptor	External number	Internal number
O4	100	512
O4	␣␣77	63
O4.2	4444	2340
O4	4444	2340

Output

When the O format code is used for output, the next item in the output list will be transferred to the external field as an octal number occupying *w* character positions in the current record; the number will be right justified by inserting spaces on the left if necessary.

The O*w*.*m* edit descriptor has a simliar effect except that the octal number will consist of at least *m* digits and, if necessary, will have leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the output item is zero then the output field will consist only of space characters.

If the value to be output exceeds *w* characters, the output field will be filled with asterisks.

The following examples demonstrate the effect of the O conversion code on output; the symbol '␣' represents a space.

Edit descriptor	Internal number	External number
O5	8	␣␣␣10
O5.4	8	␣0010
O5.0	0	␣␣␣␣␣
O5	32768	*****

15.3.1.10 The H Format Code and Character Data

Edit descriptors using the H format code are used to transfer character strings between the format specification and the current record. They do not involve program variables. They have the form:

nHstring
'string'

where:

n is the width of the character string.

string is the character string to be transferred.

If character data within apostrophes contains an apostrophe, that apostrophe must be represented by two apostrophes, for example, 'DON''T' and 5HDON'T are equivalent.

Both forms can be used in format specifications. For example, the following two formats are equivalent:

```
100   FORMAT(' ANNUAL TOTAL')
100   FORMAT(13H ANNUAL TOTAL)
```

Input

Edit descriptors using apostrophes or the H format code may not be used on input.

Output

When used for output, this edit descriptor will cause the *n* characters of *string* to be output as part of the current record. For example, either of the examples quoted above would cause

```
ANNUAL TOTAL
```

to be written to the output stream.

The ANSI Standard describes the first character of a formatted output record as a print control character which controls the vertical spacing of the record if it is sent to a display device such as a printer or screen. The table below describes the effect of various values for the print control character.

Print control character	Effect
␣	Feed one line before printing
0	Feed two lines before printing
1	Feed to head of a new page
+	No line feed
any other	As for space character

Parallel Fortran however follows the common convention that no significance is attached to the first character of a formatted record. Instead, the program **fpr** is provided (see chapter 25), and this may be used to convert output which includes carriage control characters for printing.

A convenient form of specifying the print control character is to use an edit descriptor of the following form at the beginning of a format specification

```
1Hx
'x'
```

where *x* may be one of the characters in the table given above.

15.3.1.11 The X Format Code

The X format code is used to skip characters in the input or output record. The edit descriptor has the form:

*n*X

where *n* is an integer giving the number of characters to be skipped. The X code is not concerned with transfer of data to or from a variable or array element in store.

Input

When this format code is used for input, the edit descriptor will cause *n* characters on the external record to be skipped.

Output

When this format code is used for output the edit descriptor will cause the next character that is to be transmitted to the record, to be written at a position *n* characters forward from the current position. Any unfilled positions will be filled with spaces.

15.3.1.12 The T Format Codes

The T format codes are used to specify the position in the record where the transfer of data is to begin. Their use may result in the overwriting of data already in the record. The edit descriptor takes one of the forms:

T*c*
TL*c*
TR*c*

where *c* specifies the character position at which the transfer should begin.

The T*c* edit descriptor indicates that the next character is to be transferred to or from the *c*th character position within the record, counting the first character position in the record as position one.

The TL*c* edit descriptor indicates that the next character to or from the current record is to be *c* character positions backward from the current position. If the current position is less than or equal to position *c*, then the transmission of the next character to or from the record would occur at position one.

The TR*c* edit descriptor indicates that the next character to or from the current record is to be *c* character positions forward from the current position. Note that on output, T format codes do not in themselves cause any characters to be transferred and therefore do not affect the length of the record. However if characters are subsequently written beyond any unfilled positions, then those positions will be filled with spaces.

Examples

The following format specification:

```
100   FORMAT(T16,'OF',T1'␣THE',T19,'FILEX',T6,9HBEGINNING)
```

causes the following line to be written to the output stream:

```
      THE BEGINNING OF FILEX
```

The format specification:

```
200   FORMAT(T19,'ING',TL8,'␣EDIT',T6,'SM',TL8,'␣TRAN',TR2,'ISSION')
```

causes the following line to be written to the output stream

```
      TRANSMISSION EDITING
```

15.3.1.13 The S Format Codes

The S format codes control the outputting of plus characters in numeric output fields. The edit descriptors take one of the forms

```
SP
SS
S
```

If an SP edit descriptor occurs in a format specification then a plus sign will be produced in any subsequent position which normally contains an optional plus.

If an SS edit descriptor occurs in a format specification then a plus sign will not be produced in any subsequent position which normally contains an optional plus.

If an S edit descriptor occurs then the option is restored to the compiler default, which in Parallel Fortran is the same as for the SS edit descriptor.

15.3.1.14 The B Format Codes

The B format codes control the interpretation of spaces other than leading spaces in numeric input fields. The edit descriptors take one of the following forms:

```
BN
BZ
```

If a BN edit descriptor occurs in a format specification all spaces in succeeding numeric input fields are ignored. The effect of ignoring spaces within numeric spaces is to treat the input field as if all spaces were left justified. A field of all spaces has the value zero.

The following example

```
      READ(5,10) I,J,K
10    FORMAT(BN,2I6)
```

with data

```
␣␣2␣34␣1␣2␣3␣␣␣␣␣␣
```

would cause the following values to be assigned:

- `I` to be assigned the value 234
- `J` to be assigned the value 123
- `K` to be assigned the value 0

If a `BZ` edit descriptor occurs in a format specification all such space characters in succeeding numeric input fields are treated as zero. In the example above with `BZ` specified:

- `I` would be assigned 2034
- `J` would be assigned 10203
- `K` would be assigned 0

The `BN` and `BZ` edit descriptors have no effect on output. On input they affect only `D`, `E`, `F`, `G`, `I`, and in Parallel Fortran, `O` and `Z` editing.

15.3.1.15 The Dollar ($) Format Code

This edit descriptor is used to suppress the output of a carriage return at the end of a line of output. The edit descriptor takes the form

```
$
```

The ANSI Standard does not allow the dollar edit descriptor, but it may be used in Parallel Fortran as means of generating a prompt for input.

Input

The dollar format code has no effect on input.

Output

When output is directed to a display device such as a video screen each record normally appears on a line by itself. The dollar format code may be used to suppress the carriage return such that any typed input at the screen will directly follow the output on the same line. This gives the effect of a prompt for input.

Example

```
      CHARACTER*10 OPTION
      WRITE(6,36)
36    FORMAT('ENTER OPTION: ',$)
      READ(5,40) OPTION
40    FORMAT(A)
```

will output the message on the screen in the form

```
ENTER OPTION:
```

The reply (say LIST) will then appear on the same line, thus:

```
ENTER OPTION: LIST
```

15.3.1.16 The Q Format Code

In Parallel Fortran, the Q edit descriptor is used to obtain the number of characters in the input record remaining to be transferred during a read operation. The edit descriptor takes the following form.

```
Q
```

The ANSI standard does not allow the Q edit descriptor.

Input

The input list item corresponding to the Q edit descriptor must be of integer or logical data type. The Q edit descriptor could be used to read a variable length formatted record.

For example, consider the following.

```
      READ (1,100) X,K,NCHRS,(ICHR(I),I=1,NCHRS)
100   FORMAT(F10.4,I4,Q,80A1)
```

Two fields are first read into the variables X and K. The number of characters remaining in the record is stored in NCHRS and this value is then used to control the number of characters read into the array ICHR.

Output

In an output statement the only effect of the Q edit descriptor is to skip the corresponding output list item.

15.3.2 Colon Editing

When a colon is encountered in a format specification and there are no more items in the input/output list then format control is terminated. If there are more items in the input/output list then the colon has no effect.

The example below demonstrates the effect the colon edit descriptor has on output:

```
      I= 12
      J= 24
      WRITE(6,10) I
      WRITE(6,10) I,J
      WRITE(6,20) I
10    FORMAT(' FIRST VALUE IS',I3,:,', SECOND VALUE IS',I3)
20    FORMAT(' FIRST VALUE IS',I3,', SECOND VALUE IS',I4)
```

would print the following output

```
FIRST VALUE IS 12
FIRST VALUE IS 12, SECOND VALUE IS 24
FIRST VALUE IS 12, SECOND VALUE IS
```

15.3.3 Default Field Widths

You can specify some edit descriptors without giving their field widths. In this case, the field widths are decided by the compiler on the basis of the data type of the corresponding element in the I/O list. The way in which field widths are specified is discussed in section 15.3. The default values for the *w*, *d* and *e* parts of the field descriptors are shown below.

Descriptor	Data type of list element	*w*	*d*	*e*
I, O, Z	BYTE	7		
I, O, Z	INTEGER, LOGICAL	12		
O, Z	REAL	12		
O, Z	REAL*8	23		
L	LOGICAL	2		
F, E, G, D	REAL, COMPLEX	15	7	2
F, E, G, D	REAL*8, COMPLEX*16	25	16	3
A	LOGICAL, INTEGER	4		
A	REAL, COMPLEX	4		
A	REAL*8, COMPLEX*16	8		
A	CHARACTER*n	n		

15.4 Examples of Format Specification

1. A READ statement (see chapter 16) could refer to the format statement

```
16    FORMAT(2E5.3)
```

and specify an input list which consists of two real variables X and Y. In this case, if the first ten characters in the record read by the READ statement were

```
1234599.90
```

the variable **X** will be assigned the value 12.345 and **Y** the value 99.9.

If the record also contains further fields, then the data in these fields will not be used. If the **READ** statement is executed again, another record will be read and **X** will be assigned the value corresponding to the first five characters and **Y** the value of the next five characters.

2. The **CHARACTER*8** array **AX(4)** holds the following characters:

```
AX(1)     ( 3 P G 1 1 . 4
AX(2)     , 0 P E 7 . 3 ,
AX(3)     3 ( / E 7 . 3 )
AX(4)     ) X 3 / ) ) . ,
```

The first three elements and the first character of the fourth element form a format specification; the final seven characters of the fourth element are irrelevant. A reference in a **READ** statement to the array name **AX** is equivalent to a reference to statement label **101** where statement **101** reads as follows:

```
101   FORMAT(3PG11.4,0PE7.3,3(/E7.3))
```

If the **READ** statement contains a list giving the following names:

```
A, B, C, D, E
```

in that order, where all the names are those of real variables, the first eleven characters of the next record (say the nth record of a data) will be read according to the **G** conversion code with a scale factor of 3, and the value assigned to variable **A**. The next seven characters of the record will be read according to the **E** conversion code with a scale factor of zero and the value assigned to variable **B**. The first seven characters of each of the $(n + 1)$th, $(n + 2)$th and $(n + 3)$th records will be read according to the **E** conversion code with a scale factor of zero and the values assigned to variables **C**, **D** and **E** respectively.

3. The **FORMAT** statement

```
11    FORMAT(G11.4,2(E7.3,E8.5))
```

is used in conjunction with a **WRITE** statement which lists the following real variables:

```
A, B, C, D, E, F, G, H
```

in that order. The output records contain the values of the following variables:

Record	Characters	Variable	Format
1	1 to 11	A	G11.4
	12 to 18	B	E7.3
	19 to 26	C	E8.5
	27 to 33	D	E7.3
	34 to 41	E	E8.5
2	1 to 7	F	E7.3
	8 to 15	G	E8.5
	16 to 22	H	E7.3

As the output list is not exhausted when the format specification has been completely scanned, rescanning takes place as described in section 15.2.3.

Chapter 16

Input and Output

This chapter describes the input and output facilities available.

Important Notes

- None of the facilities described in this chapter are available to programs which are linked with the standalone run-time libraries, `safrtlt4.bin` and `safrtlt8.bin` (see section 5.3). If you attempt to link programs which use these facilities with the standalone libraries, the linker will report errors.
- If more than one thread concurrently makes use of the facilities described in this chapter, they must protect the run-time library from corruption by using the `F77_THREAD_USE_RTL` and `F77_THREAD_FREE_RTL` subroutines. See section 18.2.3.

16.1 Introduction

Reading data into and writing data out from main store is controlled by *input/output statements.*

The general form of these statements is discussed in section 16.2 where references to details of each statement can be found.

Each external file is identified in an input/output statement by a unique number, a *unit number*, which takes the form of an unsigned integer.

The identification of internal files is described in sections 16.3.1.1 and 16.7 below.

Items of data for input or output are grouped into *records* which can be either *formatted* or *unformatted*.

The records within an external file are either all unformatted or they are all formatted. These two types are described in more detail in sections 16.1.1.1 and 16.1.1.2 below and the formats of input/output statements for each are referenced in those sections.

Each input/output operation begins at the start of a record. The method of access to a file may be either *sequential* or *direct* depending on the type of input/output device used. These methods of access are described in sections 16.1.2.1 and 16.1.2.2 respectively and the input/output statements available for each method are described in sections 16.3 and 16.4 respectively.

16.1.1 Format of Records

16.1.1.1 Unformatted Records

Unformatted records are input and output under the control of a `READ` or `WRITE` statement with no associated format specification. The records are representations of the internal machine form. Unformatted records will normally be output by the computer and used subsequently for re-input rather than for examination by the programmer.

Details of input/output statements for unformatted records are found, for sequential access, in section 16.3.1.2, and, for direct access, in section 16.4.3.

16.1.1.2 Formatted Records

Formatted records are input and output under the control of a **READ**, **WRITE**, **PRINT**, **TYPE** or **ACCEPT** statement in association with list-directed input/output, namelist-directed input/output or a format specification. The records are character representations of the internal values.

Details of input/output statements for formatted records are found in section 16.3.1.1 for sequential access, and in section 16.4.2 for direct access. Section 16.5 describes list-directed input and output, and section 16.6 describes namelist-directed input and output.

16.1.2 Accessing Records

16.1.2.1 Sequential Access

In sequential access, each record is read or written in sequence starting with the first record on the file.

Records on any type of input/output device may be accessed sequentially. Records on some types of device must be accessed sequentially, for example, records on a display screen. However, on certain types of input/output device such as disk it is possible to space backward past one or more records or to position at the first record on the device. Input and output using sequential access is described in section 16.3.

16.1.2.2 Direct Access

In direct access, any record can be accessed directly in an order chosen by the user.

It is only possible to use direct access on certain types of input/output devices known as direct access devices. A typical direct access device is a magnetic disk. Input and output using direct access is described in section 16.4.

16.2 Input/Output Statements

The most important input/output statements are the **READ** and **WRITE** statements.

The **READ** statement has the effect of making values from external data records available to the program by assigning them to specified variables and array elements. The **WRITE** statement has the effect of forming external records from the values of specified variables and array elements.

The **READ** and **WRITE** statements can take various forms depending on the kind of record to be handled, the kind of file on which it is held, and the facilities used to control the handling of the record.

The most general forms of the **READ** and **WRITE** statements are:

READ(*parameters*) *list*
WRITE(*parameters*) *list*

where:

parameters is a list of parameters which varies according to the kind of record being handled and the file being used.

list is an input/output list, which specifies the names of the items to have their values input or output (see

section 16.2.1). The *list* may be used in either formatted or unformatted **READ** or **WRITE** statements (see section 16.2.2 for details of the correspondence between list items and the format specification).

The effects of the various **READ** and **WRITE** statements are described in sections 16.3 to 16.5.

The **PRINT** statement and, in Parallel Fortran, the **TYPE** and **ACCEPT** statements can be used in place of the **READ** and **WRITE** statements in certain contexts.

There are three input/output statements known as *auxiliary* input/output statements, which may be used to describe, terminate, and inspect a connection between a unit number and an external file. The statements are:

```
OPEN
CLOSE
INQUIRE
```

The auxiliary input/output statements are described in section 16.8.

Other input/output statements, known as *file positioning* input/output statements, are available for limited forms of input/output device. The following statements are available for certain sequential access input/output devices:

```
ENDFILE
REWIND
BACKSPACE
```

They are described in detail in section 16.3.1.2.

Other statements, called *list-directed statements* (see section 16.5) are available. In these statements the format of the input or output data need not be specified.

16.2.1 Input/Output Lists

An *input/output* list is normally required in a **READ** or **WRITE** statement. The list has the form:

$$item_1, item_2, \ldots, item_n$$

where each $item_i$ can be the name of a variable, array, array element or character substring, or an implied **DO**-loop (see section 16.2.3). In addition, in a **WRITE** statement, an item may be any other expression except for character expressions which include elements of assumed size which are not symbolic constants. Any item or series of items may be enclosed in parentheses. An array name in an input/output list represents the whole array and thus corresponds to n separate items in the input/output list, where n is the total number of elements in the array taken in their order of storage (see section 10.1.2). Note that the name of an assumed-size dummy array must not appear in an input/output list, nor may an input/output list contain a reference to a function if the function executes an input/output statement.

An input/output list is normally required with all **READ**, **WRITE**, **PRINT**, **TYPE** and **ACCEPT** statements, whether the data is to be transferred formatted or unformatted and whether the access method is sequential or direct. If no input/output list is specified then in general one record will be skipped on input and an empty record will be written on output. However, if there is no input/output list but there is a format specification, the actions required by any non-repeatable edit descriptors will be performed until either a repeatable edit descriptor or the end of the format specification is encountered.

16.2.2 Correspondence Between Input/Output Lists and Format Codes

When a statement with a format specification and an input/output list is executed, successive items in the list are transmitted according

to the successive format codes. Where the format code is of a specific type, for example, the I format code for integer values and the L format code for logical values, then the corresponding item in the input/output list must be of the same type.

If, in an input/output statement, there are more items than format codes, then a new record is started and control is transferred to one of the following:

- If there is a group format specification: to the group format specification repeat count that is terminated by the penultimate right hand parenthesis of the **FORMAT** statement;
- If there are no group format specifications: to the beginning of the **FORMAT** statement.

The same series of format codes is then used for the next items in the input/output list.

16.2.3 Implied DO-Loops

An implied DO-loop is a series of list elements, usually array elements, that is to be repeated for different values of a DO-variable. An implied DO-loop is used to simplify the specification of array elements required in input/output operations.

It takes the form:

$$(e_1, e_2, \ldots, e_n, i=m_1, m_2, m_3)$$

where:

each e_i is a list element as defined in section 16.2.1. An e_i may be another implied DO-loop.

i is the DO-variable.

m_1 is the initial parameter.

m_2 is the terminal parameter.

m_3 is the incrementation parameter, which may be omitted, in which case it is assumed to have the value 1.

The DO-variable and parameters are analogous to those of a DO statement (see section 13.3.1). As with DO statements, implied DO-loops may be nested.

i may be the name of an integer, real or double precision variable; m_1, m_2 and m_3 may be any integer, real or double precision expression except that any functions referenced must not themselves carry out input or output operations.

The effect of the implied DO-loop is the same as if the list $e_1,e_2,\ldots,e_n$ had been written down once for each iteration of the implied DO-loop with appropriate substitution of values for any occurrence of the DO-variable i.

For input lists, i, m_1, m_2 and m_3 must not appear within the implied DO-loops except in subscripts to array names.

If one e_i in an implied DO-loop is a variable rather than an array then, on output, the same value will be output several times, and on input, several values will be assigned successively to the same variable, each value overwriting the previous value.

Example 1

A simple implied DO-loop of the form

```
(A(I),I = -1,10,1)
```

would have the same effect as the input/output list

```
A(-1),A(0),A(1),...,A(10)
```

Example 2

An implied DO-loop of the form

```
N,(A(I),B(I),I = 1,N),ALPHA(2)
```

transfers the data in the following order:

```
N,A(1),B(1),A(2),B(2),...,A(N),B(N),ALPHA(2)
```

Note that, in this example, **N** appears in the same input/output list as an implied DO-loop using it for indexing information. It also shows a specific array element, **ALPHA(2)**, appearing in the input/output list.

Example 3

The input/output list

```
A,M,MOD,((CAB(J,L),B(L), L = 1,N),J = 1,35,2)
```

causes the variables to be accessed in the following order:

```
A,M,MOD,CAB(1,1),B(1),CAB(1,2),B(2),...,CAB(1,N),
B(N),CAB(3,1),B(1),...,CAB(35,N-1),B(N-1), CAB(35,N),B(N)
```

Note that because of the position of array B in the nested implied DO-loops, every element of B is accessed a total of 18 times.

Example 4

```
(I,I = 1,10)
```

If used in a WRITE statement this implied DO-loop would output integer numbers 1,2,...,10. However, this list would be invalid in a READ statement.

16.3 Sequential Access Input and Output

Reading from and writing to sequential access input/output devices are carried out by READ, WRITE, PRINT, TYPE and ACCEPT statements. The form of these statements is described below and the use of

these statements for formatted and unformatted data is described in sections 16.3.1.1 and 16.3.1.2 respectively. Other sequential input/output statements are described in section 16.3.2.

16.3.1 READ and WRITE Statements

The basic forms of these statements for sequential access are as follows:

```
READ(k,f) list
WRITE(k,f) list
```

where:

k	is a unit specification.
f	is a format specification.
list	is an input/output list as described in section 16.2.1.

The unit specification *k* is normally an unsigned integer or integer expression which defines the unit to be used in the input/output operation. The unit would be associated with a file or device either by an **OPEN** statement (see section 16.8) or by the preconnections which exist before the program is executed. Details of preconnected units may be found in section 16.8.1.2.

The unit may also be specified as an asterisk. Such a unit identifier is associated with the primary input and output channels which are preconnected to the keyboard and the display respectively. This form of unit identifier may only be used to read or write formatted records in a sequential manner.

Section 16.3.1.1 describes other permissible forms.

The format specification *f* is normally a **FORMAT** statement label (see section 15.2.1) or a character variable or array name (see section 15.2.2). Other permissible forms are described in section 16.3.1.1.

If the records to be input or output are formatted the **READ** or **WRITE** statement must contain an *f* parameter, and if they are unformatted the statement must not contain an *f* parameter.

16.3.1.1 Formatted Sequential Access Input and Output

Input

The appropriate form of the **READ** statement for formatted sequential input is

READ(UNIT=*k***,FMT=***f***,END=***ends***,ERR=***errs***,IOSTAT=***m***)** ***list***

where:

UNIT=*k* specifies the unit number of the input/output file involved. *k* is an unsigned integer constant, variable or expression. Its value must either be zero or positive; or it may be an asterisk signifying the primary input channel (see section 16.3.1). A unit identifier may also be the name of a character variable, character array, character array element or character substring to be used in an internal file operation (see section 16.7). The characters **UNIT=** may be omitted, in which case the unit identifier must be specified first.

FMT=*f* identifies a format specification. A format identifier may be one of the following:

- The statement label of a **FORMAT** statement.
- An integer variable that has been **ASSIGN**ed the statement label of a **FORMAT** statement that appears in the same subprogram as the **READ** statement.
- A character array name, or in Parallel Fortran, any array name.

- Any character expression unless it includes elements of assumed size which are not symbolic constants.
- Any character expression that does not involve the concatenation of a character element which has a length specification of assumed size unless the character element is a symbolic constant;
- An asterisk, signifying list-directed formatting.

The characters **FMT=** may be omitted, but only if the format identifier is the second item in the list and if the first item is the unit identifier without the optional characters **UNIT=**.

END=*ends* is optional, and *ends* is the statement label to which control is transferred if an attempt is made to read data beyond the end of the file on unit number *k*.

ERR=*errs* is optional, and *errs* is the statement label to which control may be transferred when an error condition is detected.

IOSTAT=*m* is optional, and *m* is an integer variable or array element which is known as the input/output status specifier. Once the input/output statement has been completed it is assigned a value which indicates the existence of any abnormal condition encountered as follows:

- If an end of file condition is encountered, *m* is assigned a value of -1.
- If an error condition is encountered, *m* is assigned a positive value which identifies the corresponding error message.
- If no end of file condition or error condition exists, *m* is assigned a value of zero.

If the input/output status specifier is omitted, program execution will terminate if either an end of file condition is encountered and the **END=** specifier is omitted, or if an error condition is encountered and the **ERR=** specifier is omitted.

list is optional and is an input/output list.

The characters **UNIT=**, **FMT=**, **END=**, **ERR=** and **IOSTAT=** are known as *specifiers*, and if specified are not constrained to appear in the order given above. Thus for example, the input/output status specifier (**IOSTAT=**) may precede the end of file specifier (**END=**).

This **READ** statement reads in the items listed in *list* under the control of the format specification identified by *f*, from the file with unit number *k*.

An example of a formatted sequential **READ** statement is

```
READ(5,12) A,B,(C(I),I=1,10),J
```

In this example, data are read from a file with unit number 5 under control of the format specification in the format statement labelled 12. The variables **A,B,C(1),C(2),...,C(10),J** are given values in that order.

An alternative form of the **READ** statement is:

READ *f,list*

where:

f identifies a format specification as in the **READ** statement above; it may not be preceded by the optional characters **FMT=**.

list is optional, and is an input/output list. If the input/output list is omitted then the preceding comma must also be omitted.

The unit implicitly defined by this form of the READ statement is the primary input channel which is the same unit as identified by UNIT=* in the READ statement above.

Parallel Fortran allows ACCEPT as an alternative to READ in this form of the statement.

Output

The appropriate form of the WRITE statement for formatted sequential output is

```
WRITE(UNIT=k,FMT=f,ERR=errs,IOSTAT=m) list
```

where *k*, *f*, *errs*, *m* and *list* are as for the READ statement above.

The WRITE statement outputs to the unit identified by *k* all the items within *list* under the control of the format specification defined by *f*. An example of a formatted sequential WRITE statement is

```
WRITE(6,101) X,((Y(I,J),I=1,10),J=1,5)
```

Data are written to the file with unit number 6, under the control of the FORMAT specification labelled 101, in the order

```
X,Y(1,1),Y(2,1),...,Y(10,1),Y(1,2),...,Y(10,5)
```

An alternative form of the WRITE statement is:

```
PRINT f,list
```

where:

f identifies a format specification as in the WRITE statement above.

list is optional, and is an input/output list. The preceding comma must be omitted if the input/output list is not specified.

The unit identified by the **PRINT** statement is the primary output stream, and is the same as the unit identified by an asterisk in the **WRITE** statement above.

Parallel Fortran allows the use of **TYPE** in place of **PRINT**, with the same meaning.

16.3.1.2 Unformatted Sequential Access Input and Output

Input

The appropriate form of the **READ** statement for unformatted sequential input is

READ(UNIT=*k***,END=***ends***,ERR=***errs***,IOSTAT=***m***)** ***list***

where:

UNIT=*k* specifies the unit number of the external file involved. *k* is either an integer constant, integer variable or integer expression whose value is either zero or positive; it may not be an asterisk. The characters **UNIT=** are optional, and if they are omitted the unit specifier *k* must be the first item.

END=*ends* is optional, and *ends* is the statement label to which control is transferred if any attempt is made to read data beyond the end of the file on unit number *k*.

ERR=*errs* is optional, and *errs* is the statement label to which control may be transferred when an error condition is detected.

IOSTAT=*m* is optional, and specifies an input/output status specifier *m* where *m* is an integer variable or array element. After the input/output statement has been executed it

may be examined to determine whether any abnormal condition was encountered as follows:

- a value of −1 indicates that an end of file condition was encountered;
- a positive value indicates that an error condition was encountered and the value corresponds to an appropriate error message identifier;
- a value of zero indicates that no end of file condition or error condition was encountered.

If an end of file condition is detected while performing the **READ** statement and no end of file specifier (**END=**) or input/output status specifier (**IOSTAT=**) is defined, then program execution will terminate. Similarly if an error condition exists and no error specifier (**ERR=**) or input/output status specifier is defined, then program execution will also terminate. See section 17.6.4.1.

list is optional, and is an input/output list. When this **READ** statement is executed, the next record will be read and the values will be assigned in order to the variables listed in the input/output list. The number of items in the input/output list may be equal to or less than the number of values in the external record but it must not be greater than this number. If there is no input/output list, one external record is skipped.

An example of an unformatted sequential **READ** statement is as follows:

```
READ(8) X,(Y(2,K),K=1,5)
```

This statement causes data to be transferred from an input file to **X**, **Y(2,1)**, **Y(2,2)**,...,**Y(2,5)** in turn.

Output

The appropriate form of the **WRITE** statement for unformatted sequential output is

```
WRITE(UNIT=k,ERR=errs,IOSTAT=m) list
```

where *k*, *errs*, *m* and *list* are as for the **READ** statement above.

When an unformatted sequential **WRITE** statement is executed, the values of the items listed in list will be output to the file associated with the unit *k* in the order in which they occur, in internal machine form. Each unformatted **WRITE** statement will cause one, and only one, new record to be created.

The **WRITE** statement

```
WRITE(9) A,B,C
```

causes variables **A**, **B**, **C** to be written, in that order, to the file with unit number 9.

16.3.2 File Positioning Input/Output Statements

These statements are for use with magnetic input/output devices only.

16.3.2.1 The ENDFILE Statement

This statement has the following forms:

```
ENDFILE k
ENDFILE (UNIT=k,ERR=errs,IOSTAT=s)
```

where:

UNIT=*k* specifies the unit number. *k* is an integer constant or expression, or an integer variable whose value must

be zero or positive; it may not be an asterisk. The characters **UNIT=** are optional, and if they are omitted then the unit specifier must be the first item in the list.

ERR=*errs* is optional, and *errs* is the statement label to which control is transferred if an error condition is detected. *errs* is known as the error specifier.

IOSTAT=*s* is optional, and *s* is an integer variable or array element which becomes defined with a zero value if no error condition exists. If an error condition does exist it becomes defined with a positive value which identifies the corresponding error message. *s* is known as the input/output status specifier. If an error condition is detected, and if both the error specifier and the input/output status specifier are omitted, then program execution terminates.

The **ENDFILE** statement defines the end of the file associated with unit number *k* to be the current position within the file, and any data beyond the current position will be truncated. The file must be repositioned using either a **REWIND** or a **BACKSPACE** statement prior to executing a subsequent **READ** or **WRITE** on that file.

16.3.2.2 The REWIND Statement

This statement has the following forms:

```
REWIND k
REWIND(UNIT=k,ERR=errs,IOSTAT=s)
```

where *k*, *errs* and *s* are as for the **ENDFILE** statement above.

The **REWIND** statement positions the file associated with unit number *k* at the beginning of the file. The effect is that the next **READ** or **WRITE** statement referencing unit number *k* will access the first record. If the file is already positioned at the first record then this statement has no effect.

16.3.2.3 The BACKSPACE Statement

This statement has the following forms:

```
BACKSPACE k
BACKSPACE(UNIT=k,ERR=errs,IOSTAT=s)
```

where *k*, *errs* and *s* are as for the **ENDFILE** statement above.

When a **BACKSPACE** statement is executed, the effect is that the next **READ** or **WRITE** statement referencing unit number *k* will operate on the previous record of the file. If the file was positioned at its first record before the **BACKSPACE** statement is executed, then the statement will have no effect.

If the **BACKSPACE** statement occurs immediately after an **ENDFILE** statement, it has the effect of back-spacing over the end-of-file marker.

Backspacing over records which have been written using list-directed formatting (see section 16.5) is prohibited.

16.4 Direct Access Input and Output

Reading from and writing to a direct access input/output device is carried out by **READ** and **WRITE** statements of the form described below. At any time, any record may be read or written; there is no requirement to start at the first record. Writing to an output file alters only each record written, without destroying any record before or after it. Data items are not written across record boundaries, nor are they read from across record boundaries.

Data can be accessed directly only on direct access devices, usually magnetic disks.

Each record in a direct access file is assigned a number, called its record number, by which it can be referenced. The first record of

a file is numbered one and the rest are numbered consecutively in steps of one. Record numbers appear in direct access input/output statements. Note that all the records of a direct access file have the same length.

The **READ** and **WRITE** statements that are used for reading from and writing to direct access files are described in section 16.4.1 below.

16.4.1 READ and WRITE statements

The basic forms of **READ** and **WRITE** statements for direct access are as follows:

```
READ(k,REC=r) list
WRITE(k,REC=r) list
```

where:

k is an integer variable, or integer constant or integer expression and gives the unit number to be used in the input/output operation. The unit number must be zero or positive.

r is an integer expression whose value is positive. It specifies the number of the first record that is to be read or written.

list is optional and is an input/output list as described in section 16.2.1.

16.4.2 Formatted Direct Access Input and Output

Input

The appropriate form of the **READ** statement for formatted direct access input is

READ(UNIT=*k***,FMT=***f***,REC=***r***,ERR=***errs***,IOSTAT=***s***)** ***list***

where:

UNIT=*k* identifies a unit number. *k* is an integer expression whose value is zero or positive. The characters **UNIT=** may be omitted provided that the unit identifier is the first item.

FMT=*f* identifies a format specification. A format identifier may be one of the following:

- the statement label of a **FORMAT** statement.
- an integer variable that has been assigned the statement label of a **FORMAT** statement that appears in the same subprogram as the **READ** statement.
- a character array name or, in Parallel Fortran, any array name.
- any character expression unless it includes an element which has an assumed size and is not a symbolic constant.

The characters **FMT=** may be omitted, but only if the format identifier is the second item in the list and if the unit identifier is the first item without the optional characters **UNIT=**.

REC=*r* is an integer expression whose value must be positive. It represents the record number of the first record which is to be read.

ERR=*errs* is optional and *errs* is the label of the statement to which control may be transferred when an error condition is encountered while executing the input/output statement.

IOSTAT=*s* is optional, and specifies an input/output status specifier. *s* is an integer variable or array element which becomes defined with a zero or positive value when the input/output statement has been executed. If no error condition exists then a value of zero is assigned, otherwise the value assigned is the number of the error message which corresponds to the error detected.

If the input/output status specifier and the error specifier are both omitted, program execution will terminate when an error condition is encountered; see section 17.6.4.1.

list is optional and is an input/output list as defined in section 16.2.1.

This **READ** statement transfers data from a file on a direct access device under control of the format specification identified by *f*, and assigns the values to the items within the input/output list. The file from which data are read must be a direct access file. It is an error to attempt to read a record from beyond the current end of the file.

Output

The appropriate form of the **WRITE** statement for formatted direct access output is

```
WRITE(UNIT=k,FMT= f,REC=r,ERR=errs,IOSTAT=s) list
```

where *k*, *f*, *r*, *errs*, *s* and *list* are as for the **READ** statement above.

This **WRITE** statement transfers data under control of the format specification *f* from items in the input/output list to a direct access file. The data are written starting at record *r*. If record *r* already exists in the file it will be overwritten. If the values in the input/output list are not sufficient to fill the record the remainder of the record is filled with spaces.

If record *r* lies beyond the current end of the file, then the file will be extended, but the contents of any records between the previous end of the file and record *r* will be undefined.

If an input/output list is not specified the only data that will be written will be any character data which may appear at the beginning of the format specification.

16.4.3 Unformatted Direct Access Input and Output

Input

The appropriate form of the **READ** statement for unformatted direct access input is:

```
READ(UNIT=k,REC=r,ERR=errs,IOSTAT=s) list
```

where *k*, *r*, *errs*, *s* and *list* are as defined in section 16.4.2.

This statement transfers data from record *r* in the direct access file associated with unit number *k* to the items in the input/output list; only one record is read and so the input/output list must not specify more values than can be contained in one record. The file from which the data are being transferred must be a direct access file.

Output

The appropriate form of the **WRITE** statement for unformatted direct access output is:

```
WRITE(UNIT=k,REC=r,ERR=errs,IOSTAT=s) list
```

where *k*, *r*, *errs*, *s* and *list* are as defined in section 16.4.2.

This statement transfers data from the items within the input/output list to record *r* in the direct file on unit *k*. If record *r* already exists in the file then it will be overwritten. The input/output list must not specify more values than can fit into a single record. If the values

specified do not fill the record, the remainder of the record becomes undefined. If the input/output list is omitted then the entire output record becomes undefined.

If record *r* lies beyond the current end of the file, then the file will be extended, but the contents of any records between the previous end of the file and record *r* will be undefined.

An example of an unformatted direct access **WRITE** statement is

```
WRITE(ERR=999,UNIT=30,REC=I+J) IARRAY,(A(I,K),K=4,8)
```

This statement will write a record to the file with unit number 30. The value of the expression **I+J** identifies the particular record within the file to which the variables **IARRAY**, **A(I,4)**, **A(I,5)**, **A(I,6)**, **A(I,7)**, **A(I,8)** are to be written. Control will be transferred to the statement labelled 999 should an error condition occur (for example if the record specifier (**REC=**) has a negative value).

16.5 List-Directed Input and Output

The use of list-directed input/output statements allows data to be read or written without the restrictions imposed by format specifications.

16.5.1 The READ Statement

The list-directed **READ** statement may take the following form:

READ(UNIT=*k***,FMT=*,END=***ends***,ERR=***errs***,IOSTAT=***s***)** *list*

where *k*, *ends*, *errs*, *s* and *list* are as defined in section 16.3.1.1.

Execution of the **READ** statement inputs values from external records and assigns them, in order, to the items in the input list. In the case of an array name the elements are given values in order of

storage (that is, with the leftmost subscript expression varying most rapidly). Each value input from an external record should be terminated by a value separator which may be one of the following:

- a comma optionally preceded by one or more spaces and optionally followed by one or more spaces;
- a slash optionally preceded by one or more spaces and optionally followed by one or more spaces;
- one or more spaces between two values or following the last value.

The input operation is terminated by the satisfaction of the input list or by the reading of a slash.

Items of the input list will not be assigned values if they either correspond to null data items (see below) or if a slash is specified in the data before their values are read. Such items in the input list will retain any value they held prior to the **READ** statement.

The type of each item from the input list must correspond with the form of data from the external medium. However Parallel Fortran allows character constants to be assigned to non-character items in the input list.

Each **READ** statement starts with a new record, and reads as many records as are necessary to provide data to satisfy the input/output list.

An alternative form of the list-directed **READ** statement is:

READ *, *list*

where *list* is as described in section 16.3.1.1. In this case, input will be taken from standard input.

Parallel Fortran allows **ACCEPT** as an alternative to **READ** in this form of the statement.

16.5.2 Input Data

When a list-directed **READ** statement is executed, reading begins at the start of the next unaccessed record in the input file and continues until either each item in the input list has been given a value or a slash (**/**) is encountered in the input. Any data in the last record accessed by a **READ** statement which follows a slash or is not required for input cannot be accessed. Any input list items not given a value before a slash is reached retain their current value (or remain undefined).

The input stream consists of a series of data items which are associated in their order of occurrence with the items of the input list. Data items are separated by one or more spaces or by a single comma optionally preceded and optionally followed by spaces. Note that the end of a record has the effect of a space, except when it appears within a character constant (see below). An item may be:

A numeric constant: this may take any of the forms listed in section 9.2.1 apart from Hollerith, hexadecimal, octal or binary constants. Numeric constants may not contain any embedded spaces except between the parts of a complex constant, in which case any number of spaces is permissible. The end of a record may not appear within a constant unless the constant is a complex value, in which case the end of record may occur between the real part and the comma or between the comma and the imaginary part.

The type of the constant must be the same as that of the corresponding list item, but there need be no correspondence of length.

A logical constant: if the corresponding item is of type logical, the data item may be any value acceptable to L editing (see section 15.3.1.6). However commas or slashes are not permitted as optional characters.

A character constant: a data item may be a non-empty string of

characters enclosed within apostrophes. Note that the form *n*H... is not permitted. Each apostrophe that is part of the character value must be represented by two consecutive apostrophes. The constant may be continued on as many records as needed and an end of record does not cause a space or any other character to become a part of the value. Double quote characters as described in section 9.2.1.11 have no significance in list-directed input.

The corresponding input list item need not be of type character, and there need be no correspondence of length. Note that the constant is assigned in the same manner as if the constant appeared in a character assignment statement (see section 12.3).

A null item: a null item may take one of the following forms:

- No characters appearing between two successive value separators.
- No characters preceding the first value separator in the first record input by a **READ** statement.

The value (or undefined status) of the corresponding list item is left unchanged.

A repeated item: any of the above items may be preceded by a positive unsigned integer constant and an asterisk (*n**). *n** must not contain any embedded spaces and may not extend over a record. A repeated null item occurs if the next character after * is a value separator.

The effect of a repeated item is that the next *n* items from the input list have the same value read into them; they must all have the same type as the value. If a null item is repeated, the next *n* items from the input list are left unchanged.

A constant is terminated by the first space, end of record, or comma after its syntactic completion (that is, after a closing bracket for a complex constant or the closing apostrophe for a character constant).

Note that spaces are never used as zeros, and all spaces are considered to be part of some value separator except in the following circumstances:

- spaces embedded within a character constant.
- spaces which precede or follow the real or imaginary part of a complex constant.
- leading spaces in the first record input by a **READ** statement unless they are immediately followed by a comma or a slash.

Example

If the next two records read from a file are

```
1056␣198765,␣␣␣␣␣,␣␣␣␣,␣␣␣␣50*␣␣␣␣␣␣␣␣␣␣
50*(4E1,5E1)␣␣␣T,'ONE''S'␣'TWO'/␣␣456␣␣
```

and the file is accessed by the statement

```
READ(15,*) I,J,K,L,(A(M),M=1,50),(B(M),M=1,50),P,X,Y,N
```

I is given the value 1056, J the value 198765, K, L and A are unchanged, the first 50 elements of B are each given the complex value $40 + 50i$, P the value .TRUE., X the value ONE'S and Y the value TWO. N remains unchanged and the value 456 is not accessed since the slash intervenes.

16.5.3 Output Statements

The list-directed forms of the WRITE, PRINT and TYPE statements output the values associated with the items in an output list. The values are written to a device or file in a form which is compatible with the type of the item.

16.5.3.1 The WRITE statement

This statement has the form:

```
WRITE(UNIT=k,FMT=*,ERR=errs,IOSTAT=s) list
```

where *k*, *errs*, *s* and *list* are as for the **READ** statement above.

The **WRITE** statement outputs the values of each item in the output list to a file identified by unit number *k*.

16.5.3.2 The PRINT and TYPE Statements

The **PRINT** statement has the form:

```
PRINT *,list
```

where *list* is an input/output list as defined in section 16.2.1.

The **PRINT** statement writes data to be written to the primary output channel which is usually connected to the screen. The unit will be the same as if an asterisk had been specified in a list-directed **WRITE** statement. The following is an example of a list-directed **PRINT** statement:

```
PRINT *,I,J,K,(A(I),I = 1,100)
```

Parallel Fortran allows the use of **TYPE** in place of **PRINT**, with the same meaning.

16.5.4 Output Data

When a list-directed **WRITE**, **PRINT** or **TYPE** statement is executed, the values of all elements of each list item are output in sequence. Each record starts with a single space and contains at least one space between each value output and no embedded spaces within items (other than spaces within character values). A record may end with no spaces or with one or more spaces.

The forms of output are as follows:

For integer values: all digits are output except for leading zeros. If negative, the value is preceded by a minus sign.

For real values: all significant digits are output. If the value to be output contains d significant digits, and the value is greater than or equal to 0.1 and less than 10^d, the number is output in a form which is similar to the effect of using an F edit descriptor (see section 15.3.1.2) with a zero scale factor, that is, without an exponent; otherwise, the number is output with an exponent in a form that is similar to the effect of using an E edit descriptor (see section 15.3.1.3) with a scale factor of 1. The value is preceded by a minus sign if it is negative.

For complex values: an opening parenthesis is output followed by the value of the real part, followed by a comma, followed by the imaginary part, followed by a closing parenthesis. The real and imaginary parts are output as for real values.

For logical values: the single character T or F is output.

For character values: all the characters are output without spaces preceding or following the characters other than any spaces that may be part of the character value. Character values that are output are not delimited by apostrophes.

As many records as are necessary will be written but the end of a record will not occur within a value, apart from a complex or character value. The end of a record may appear within a complex value between the comma and the imaginary part only if the entire constant is as long as, or longer than an entire record. Character values will be extended across as many records as required and each such new record will have a space character inserted at the beginning for carriage control (see section 15.3.1.10).

Note that slashes, as value separators, and null items are not produced by list directed formatting.

The ANSI Standard permits repeated items of the form $n*$ (see section 16.5.2) to be output; however, this form is not used by Parallel Fortran.

16.6 Namelist-Directed Input and Output

Namelist-directed I/O enables the programmer to input or output a group of variables with a single statement. This group of variables is called a *namelist*. A namelist has a symbolic name, which is defined by a **NAMELIST** statement. Whenever an input or output statement refers to the namelist's name, the whole group of variables is transferred, in a special format which includes their names.

Namelist-directed I/O is an extension to the ANSI standard.

16.6.1 The NAMELIST statement

The **NAMELIST** statement has the following format.

NAMELIST /$name_1$/a_1,a_2,.../$name_2$/b_1,b_2,...

where:

$name_1$,$name_2$,...
: are the names of namelists.

a_1,a_2,...,b_1,b_2,...
: are the lists of variables or array names which are to form the namelists in question.

The **NAMELIST** statement groups together under one name the variables and arrays whose names are specified in the statement, so that they can be input or output by a single namelist-directed I/O statement. A namelist must be declared by a **NAMELIST** statement before it is used, and it must be declared only once.

The variables and arrays in the namelist may be of any type. A variable or array may appear in more than one namelist. The following things, however, may not appear in namelists:

- Array elements;
- Character substrings;
- Dummy arguments.

However, values may be given to array elements and character substrings by a namelist-directed input statement, provided that the array or character variable of which they are a part is included in the appropriate namelist.

For example, the following **NAMELIST** statement:

```
NAMELIST /NAM1/Q,B,I,L,J,K /NAM2/C,J,I,L,K
```

This defines two namelists, **NAM1** and **NAM2**. The variables `Q` and `B` belong to namelist **NAM1**, and `C` belongs to namelist **NAM2**. `I`, `J`, `K` and `L` belong to both namelists, although in different orders.

16.6.2 Input Statements

The namelist-directed input statement takes one of the following forms:

```
READ (UNIT=k,NML=nml,END=ends,ERR=errs,IOSTAT=m)
READ nml
ACCEPT nml
```

where:

k, *ends*, *errs* and *m*
: are as defined in section 16.3.1.1, except that *k* cannot be an internal file.

nml
: is the name of a namelist, already defined by a **NAMELIST** statement.

16.6.3 Input Data

Data Blocks

The *data block* required by a namelist-directed input statement consists of a number of *items*, separated by *item separators*. An item separator is a comma or a sequence of one or more spaces; extra spaces may be inserted on either side of a comma. Except where noted below, items may not include embedded spaces. The items are as follows.

- The first item consists of an ampersand '**&**' or dollar '**$**' followed by the name of the namelist.
- After this there follows a sequence of *data items*. These are discussed below.
- Finally, the data block is terminated by an item consisting of an ampersand '**&**' or dollar '**$**' followed by **END**.

Data Items

The format of a data item is as follows.

entity=value

where:

entity is the name of an array, an array element, substring, array element substring or a variable.

value is the value to be assigned to *entity*.

There may be a sequence of one or more spaces on either side of the '='. Array elements and substrings are specified in the usual Fortran way; extra spaces may be inserted within the parentheses. Subscripts must be constants.

The *value* may be one of the following.

A constant: constants follow the usual Fortran formats, and may be of type integer, real, character, complex or logical. Symbolic (**PARAMETER**) constants cannot be used.

Numeric constants may not be Hollerith, hexadecimal, octal or binary. Integer values may be supplied for real variables, but not vice versa. Extra spaces may be inserted within the parentheses of a complex constant.

The allowed values for logical constants are those which are acceptable to **L** editing (see section 15.3.1.6).

Character constants are enclosed in apostrophes (**'**); apostrophes within the constant are represented by two apostrophes. Character constants enclosed in double quote characters (**"**) are not allowed. Character constants may be assigned to numeric variables.

A list of constants: this may be used when the *entity* is an array. The constants in the list are separated by item separators (see above). A sequence of identical constants may be represented in the form *n*c*, where *n* is a constant unsigned integer repetition count, and *c* is the constant value. Values are assigned to the array starting at its lowest-numbered element; the number of elements listed must not be greater than the size of the array.

A null value: A null value is represented by two successive commas in a list, an initial comma or a trailing comma. A sequence of null values can be represented by the form *n**, where *n* is a constant unsigned integer repetition count. The effect of a null value is that the value of the object to which it is assigned is unchanged.

Data Blocks and Records

As many records as are needed may be used to hold a data block. The first column of each record is unused, and must be blank. A new record may be started anywhere in the data block where a space is

allowed: a record boundary is the equivalent of a space, and so by itself can constitute an item separator. A record boundary within a character constant, however, is *not* treated as a space, but is entirely ignored.

The *entity* of a data item must be contained in a single record, including any subscripts or substring specifiers. There may not be any space before the first item: that is, the ampersand '**&**' or dollar '**$**' must be in column 2 of the record.

Evaluation of the Data

When a namelist-directed input statement is executed, Fortran reads records from the specified unit until a record is encountered which starts a data block for the appropriate namelist. If such a record is not found, an end-of-file is signalled.

Fortran then reads the data items in the data block, and assigns the values to the specified entities. The order of the data items is not significant. Only entities named in the corresponding **NAMELIST** statement may be assigned values in the data block; any others will cause an error.

Entities which are not assigned values, or which are assigned null values, will have unchanged values. The same applies to elements of an array which are not assigned values, and the remaining parts of a character variable when a substring is assigned a value.

16.6.4 Output Statements

The namelist-directed output statement takes one of the following forms:

```
WRITE (UNIT=k,NML=nml,ERR=errs,IOSTAT=m)
PRINT nml
TYPE nml
```

where:

k, *errs* and *m*
: are as defined in section 16.3.1.1, except that *k* cannot be an internal file.

nml
: is the name of a namelist, already defined by a **NAMELIST** statement.

16.6.5 Output Data

When a namelist-directed output statement is executed, the data are output in the form of a data block which can be read using namelist-directed input.

All variable and array names specified in the namelist and their values are written out according to their type. The fields for the data are made sufficiently large to contain all the significant digits. The values of a complete array are written out in columns.

16.6.6 Example of Namelist-Directed I/O

Consider the following program.

```
REAL  A(3)
INTEGER  I(3,3), L(3,3)
DATA A/3*0.0/, I/9*0/, L/9*1/
NAMELIST/NAM1/A,B,I,J,L/NAM2/C,I,J,L
READ(5,NAM1)
C=428000
WRITE(7,NAM2)
```

Suppose that this is executed with the following input data (note that the data start in the second column of each record).

```
␣&NAM1 I(2,3)=5,J=4,B=3.2
␣A(3)=4.0,L=2,3,7*4,&END
```

The **NAMELIST** statement defines two namelists, **NAM1** and **NAM2**. The **READ** statement causes input data to be read from I/O unit 5 into the variables and arrays specified by **NAM1**, as follows.

The first data record is read and examined to verify that it is the start of a namelist data block, and that its name is consistent with the namelist specified in the READ statement. If it were not, Fortran would continue reading data records until the right one was found.

When data are read, the integer constants 5 and 4 are placed in I(2,3) and J respectively. Real constants 3.2 and 4.0 area placed in B and A(3) respectively. Then, since L is an array name not followed by a subscript the entire array L is filled with the succeeding constants. Therefore, the integer constants 2 and 3 are placed in L(1,1) and L(2,1) respectively, and the integer constant 4 is placed in L(3,1), L(1,2),...,L(3,3).

The WRITE statement causes data to be written from the variables and arrays specified by NAM2 to I/O unit number 7 as follows:

```
␣&NAM2
␣C=428000.0,
␣I= 0,  0,  0,  0,  0,  0
␣ 0,  5,  0,
␣J=3,
␣L= 2,  3,  4,  4,  4,  4
␣ 4,  4,  4,
␣&END
```

16.7 Internal Files

If the first parameter to a formatted sequential READ or WRITE statement (see section 16.3.1) is the name of a character variable, character array element, or character array, or if it is a character substring, then the input/output operation is to be carried out on an internal file consisting of that variable, array element, array, or substring.

An internal file has the following properties:

- A record of an internal file is a character variable or character array element or character substring.

- If the file is a character variable, character array element, or character substring it consists of a single record whose length is that of the variable, array element, or substring, respectively. If the file is a character array it is treated as a sequence of character array elements, each of which is a record of the file. Each record of the file has the same length, namely the defined array element length.

- If the number of characters being written to the file is less than the length of the record, the remaining portion of the record is filled with spaces.

- An internal file is always positioned on the first record at the start of a **READ** or **WRITE** statement accessing that file.

- Reading and writing records may only be performed using sequential access formatted input/output statements that do not specify list-directed formatting.

The character variable, character array element, or character array being used as an internal file must not appear in the input/output list nor contain the format being used when accessing that file.

Example

In the following code

```
      CHARACTER*16 TEXT
      :
      WRITE(TEXT,10)I
10    FORMAT('VALUE OF I =',I4)
```

if I contains the value of 136 when the **WRITE** statement is executed the effect will be equivalent to assigning the character string '**VALUE OF I = 136**' to the character variable **TEXT**.

The **ENCODE** and **DECODE** statements, which provide an alternative method for performing internal I/O, are described in appendix D.1.

16.8 Auxiliary Input/Output Statements

The input/output statements above describe the manner in which data may be transferred between internal storage and external media, and between internal storage and internal files. They also describe file positioning statements. The following sections describe auxilary input/output statements which may be used to define a connection between a unit number and a file, terminate such a connection, or interrogate the attributes of either a connection or a file.

16.8.1 Unit and File Connection

The physical association of a unit to an external file is known as a connection. Prior to program execution a connection may be predefined by the system. This is known as *preconnection*. For example, the list-directed input statement

```
READ *,list
```

makes use of the preconnection to standard input.

Internal to a program a connection may be established by means of the **OPEN** statement (see section 16.8.2)

A connection is between a unit and a file. No unit may be connected to more than one file at the same time and similarly no file may be connected to more than one unit at the same time. However, means are available to terminate a connection (see section 16.8.3), and to connect a unit to a different file. No **READ**, **WRITE**, **PRINT**, **TYPE** or **ACCEPT** statement can be executed without a connection to the specified unit.

The properties of a connection include the following:

1. The type of access, either direct access or sequential access.
2. The kind of records, either formatted or unformatted.

3. The length of the records if the file is to be accessed with direct access input/output statements.

File existence is totally independent of a connection; that is, a file may be connected but not exist (see section 16.8.1.2).

16.8.1.1 File Properties

A file property is a characteristic of an external file which exists for the life-span of the file. Taken together a file's properties describe the permissible methods that may be used to access the file.

Within the context of the ANSI Standard a file is attributed the following properties:

- The file may exist, or may not exist. If it does not exist, then it has no other property. A file is said to exist for a program if the program may transfer data either to or from the file, provided that it does not have to be created first. A new file may be created by executing an `OPEN` statement (see section 16.8.2) or by writing a record to the file if the file is preconnected.

- Its records may either be all formatted, or all unformatted. A file may not contain both types of record.

- It may be accessed with direct access input/output statements, or it may be accessed with sequential access input/output statements. Some files may be accessed with either type of statement, but note that a given connection is only for a single type of access.

- A file may have a name. If it has no name then it is a temporary file which will cease to exist after program termination.

When a connection is defined (see section 16.8.2) between a unit and a file, the properties of the connection must be compatible with the properties of the file. For example, it is not valid to define a

connection for direct access when the file to be connected is a line printer.

16.8.1.2 Preconnection

No unit may be referenced by a **READ**, **WRITE**, **PRINT**, **ACCEPT** or **TYPE** statement unless it is first associated with a file or device. This association is known as a *connection* and without a connection no data may be transferred between a file and a unit. A connection may be established:

- by the **OPEN** statement (see section 16.8.2).
- implicitly by the system before a program is executed. This is known as a *preconnection.*

A unit which is preconnected may be referenced in an input/output statement without first establishing a connection between the unit and a file. I/O units in Parallel Fortran are preconnected as follows:

Unit	Connection
*	Standard input if reading; standard output if writing
5	Standard input
6	Standard output
all others	a file with a default filename (see below)

The default filename is of the form **FORT***nnn*.**DAT** where *nnn* is the unit number. The file is assumed to be in the current directory.

If a preconnected file does not exist then it will be created when the file is first written or read. Thus, in the following example, if the first reference to unit 73 were:

```
      WRITE(73,1000) (I,I=1,3)
1000  FORMAT(I1)
```

the file **FORT073.DAT** will be created if it does not exist. Three records will be written to the file; the first record would contain the

character '1', the second record the character '2', and the last record the character '3'.

Standard input and standard output may be redirected at run time using the usual MS-DOS conventions. See section 3.4.2.

Direct Access

In Parallel Fortran the effect of using direct access I/O statements to reference a preconnected file differs from that described in section 16.4. Since the connection has not been defined by an **OPEN** statement, Fortran cannot tell what the record length of the file is; MS-DOS does not record this information. Accordingly, the file is treated as if it had a record length of 1, that is, as if it had been opened with the specifier **RECL=1**. This means that the record selector (**REC=1**) in a **READ** or **WRITE** in fact specifies a byte/character displacement from the start of the file at which the transfer of data is to begin. As much data as required to satisfy the input/output list is transferred.

For example, unit 31 might be referenced for the first time by this statement.

```
READ(31,REC=100) IARR
```

In this case, the unit would be connected to the file **FORT031.DAT** and data would be read from byte position 100 within the file into the array **IARR** until all the elements of **IARR** had been assigned.

16.8.2 The OPEN Statement

The **OPEN** statement provides a means of accessing files that are not preconnected. If a file does not exist then it may be created. The **OPEN** statement may also be used to create a new file on a preconnected unit, and to alter certain properties of a connection between a file and a unit.

Once the **OPEN** statement has established a correspondence between a given unit number and a specified file then both the unit and the file are said to be connected, and hence a **READ** or a **WRITE** statement can be executed on the unit and hence the file. Without a connection (or preconnection) a **READ** or **WRITE** statement cannot be executed.

When a file is opened, it is positioned at the first record.

The general form of the **OPEN** statement is:

OPEN(*spec*$_1$**,** *spec*$_2$**,...,** *spec*$_n$**)**

where each *spec*$_i$ is one of the following:

UNIT=*k* identifies a unit number where *k* is an integer expression whose value is either zero or positive. The characters **UNIT=** are optional and if they are omitted the unit specifier must be the first item in the list; otherwise its position in the list is not fixed.

All other specifiers may be omitted, but if they are specified they may appear anywhere within the list. The specifiers are described below together with any assumed value that may be used if a specifier is not defined.

IOSTAT=*ios* defines an input/output status specifier which may be the name of either an integer variable or integer array element. It will become defined with either a positive value or a value of zero. If an error condition exists the input/output status specifier is assigned the identifier of an error message which corresponds to the error; if no error condition is detected it is assigned the value zero.

Program execution will terminate if an error condition is detected and neither an input/output status specifier nor an error specifier (see below) is defined.

ERR=*errs* is an error specifier and defines a statement label to which control is transferred if an error condition exists. If the error specifier is omitted, and also the input/output status specifier (see above), then program execution will terminate when an error condition is detected.

FILE=*fn* specifies the name of a file to be connected to the defined unit. *fn* is a character expression whose value must represent a valid filename once any trailing spaces have been removed. If the file specifier is omitted then the value assumed by *fn* depends on whether the specified unit is connected. Should the unit be connected, then the name of the file to which it is connected is assumed; otherwise the default value used for the file specifier is dependent upon the value of the status specifier (see below).

Note that because *fn* is a character expression, the character '\' is interpreted as an escape character, as described in section 9.2.1.11. This means that if you wish to use an MS-DOS pathname when specifying the file to open, you must double the '\' character. For example:

```
OPEN (UNIT=10, FILE='C:\\PROJECTA\\DATA\\SET12.DAT')
```

NAME=*fn* is allowed in Parallel Fortran as an alternative to FILE=.

STATUS=*sta* is a status specifier. *sta* is a character expression whose value may be one of the following:

```
OLD
NEW
SCRATCH
UNKNOWN
APPEND  (in Parallel Fortran)
```

Any trailing spaces in the value are ignored. The status specifier defines the existence of the file to be connected. If **OLD** is specified the named file must exist. Conversely if **NEW** is specified the named file must not exist but it will be created by the **OPEN** statement provided no error occurs. The values **OLD** or **NEW** may only be used if a file specifier is defined; while the value **SCRATCH** may only be used if no file specifier is defined. **SCRATCH** causes the specified unit to be connected to a temporary file which exists only until either that unit is closed (see section 16.8.3) or the program terminates.

If the status specifier is omitted, the default value is **UNKNOWN**. **UNKNOWN** assumes the status of the named file if a file specifier has been defined (that is either **NEW** or **OLD**) or the status of the file to which the unit is connected if no file specifier has been defined. If there exists no connection and the file specifier is omitted then the file specifier will assume the name of the preconnected file if the status specifier has the value **UNKNOWN**.

In Parallel Fortran the status specifier may also have the value **APPEND**. **APPEND** assumes the status of a named file (that is, either **OLD** if the file exists or **NEW** if it does not). If the file exists, it will be positioned just after the last record in the file and any data which is written will be appended onto the end. The value **APPEND** may only be used if the file is to be connected for sequential access.

Once the **OPEN** statement has successfully established a connection, the status of the connected file becomes **OLD** unless the file is a temporary file.

READONLY specifies that an existing file is to be opened for reading only; attempts to write to it will cause a run-time error.

ACCESS=*acc* defines the manner in which the connection is to access the file. *acc* is a character expression whose value may be either **SEQUENTIAL** or **DIRECT**; any trailing spaces in the value will be ignored. The value **SEQUENTIAL** specifies sequential input/output and the value **DIRECT** specifies direct access input/output. When a new file is created the specified access method becomes a property of the file, that is the file is created as a sequential or direct access file; while for an existing file, the file must be capable of supporting the specified access method. A value of **SEQUENTIAL** will be assumed if the access specifier is omitted.

FORM=*fm* specifies whether the file is to be accessed with either formatted or unformatted input/output statements. *fm* is a character expression whose value when trailing spaces have been removed is either **FORMATTED** or **UNFORMATTED**. If **FORMATTED** is specified the connected file may contain no unformatted records, and if **UNFORMATTED** is specified the connected file may contain no formatted records.

Note that the type of the records is a file property. If the form specifier is omitted, a value of **UNFORMATTED** is assumed if the connection specified is for direct access, while a value of **FORMATTED** is assumed if the connection is for sequential access.

RECL=*rl* is a record length specifier. *rl* is an integer expression whose value must be positive. It specifies in units of bytes the length of each record in a file being connected for direct access. For an existing file the specified length must not be greater than the actual record length of the file. For a new file, the **OPEN** statement creates the file with a record length of *rl*. If the connection defined is for sequential access, the record length specifier must be omitted; otherwise it must be specified.

RECORDSIZE=*fn*
: is allowed in Parallel Fortran as an alternative to RECL=.

BLANK=*blk*
: may only be specified for a connection which is to be used for formatted input/output, and defines the interpretation to be applied to space characters within numeric input fields. *blk* is a character expression whose value when any trailing spaces have been removed is either **ZERO** or **NULL**. If **ZERO** is specified then all spaces in numeric input fields read from the specified unit are treated as zeros apart from leading spaces. If **NULL** is specified all spaces are ignored. A field which consists entirely of spaces always has the value zero. **NULL** is the assumed value if the blank specifier is omitted.

Each specifier in an **OPEN** statement may appear at most once. If a specifier is defined then it is not constrained to appear in the order given above. Thus for example, the blank specifier (**BLANK=**) may precede the form specifier (**FORM=**). The ANSI Standard requires that where the value of a specifier is character data then that data must be in upper case; Parallel Fortran allows such data to be specified in lower case.

A unit may be connected by an **OPEN** statement within any program unit of an executable program. Once connected the unit may be referenced in any program unit of the executable program.

16.8.2.1 Examples

Example 1

```
OPEN(UNIT=273,FILE='FIL001')
```

defines a connection between unit 273 and the file **FIL001**. In the absence of other specifiers the connection is specified for sequential

formatted input/output and any spaces which are read in numeric fields are to be ignored. As the status specifier has not been defined a value of **UNKNOWN** is assumed and the file will be created if it does not exist.

Example 2

```
OPEN(ACCESS='DIRECT',RECL=160,UNIT=10,
FILE='DFX',STATUS='OLD')
```

connects the file **DFX** to unit 10 for direct access. The file contains unformatted records each of which is 160 bytes long. The file is also assumed to exist.

Example 3

```
OPEN(1,STATUS='UNKNOWN',BLANK='ZERO')
```

either refers to an existing connection to unit 1 and changes the interpretation of spaces within formatted numeric input fields to **ZERO**; or, if no connection exists, it connects unit 1 to the file **FORT001.DAT**, which will be created if it does not exist. The connection established will only be valid for formatted input/output.

Example 4

```
OPEN(22,ACCESS='DIRECT',BLANK='NULL')
```

is not a valid statement as the record length specifier is not defined. In addition because the form specifier is omitted, the assumed access form is unformatted input/output which is incompatible with the specification of the blank specifier.

16.8.2.2 Changing the Properties of a Connection

When a unit becomes connected to a file, the same unit may appear in an **OPEN** statement to either define a new connection or to change certain properties of the current connection.

A new connection is defined when the filename specified in an **OPEN** statement is not the same file as the file to which the specified unit is already connected. The effect is as if an implicit **CLOSE** statement (see section 16.8.3) without a status specifier is executed immediately prior to the **OPEN** statement.

When the file specifier is the same as the name of the connected file then the current connection is specified. If the file is a file that is preconnected and does not exist, the values specified by the **OPEN** statement become a part of the connection and the file is also created. Otherwise, should the connected file exist then only the **BLANK=** specifier may have a value that is different from the one currently in force. Note that if the file specifier is omitted then the assumed filename is the name of the connected file unless **STATUS=SCRATCH** was specified.

A file which is already connected to a unit may not be connected to another unit unless its current connection is first terminated by a **CLOSE** statement (see section 16.8.3).

Example 1

The sequence

```
OPEN(73,FILE='DATA3',...)
⋮
OPEN(73,FILE='DATA4',...)
```

will first connect unit 73 to the file **DATA3**. At the second **OPEN** statement, unit 73 becomes connected to the file **DATA4** after first terminating the original connection.

Example 2

The sequence

```
OPEN(2000,FILE='RESULTS')
⋮
OPEN(2000,BLANK='ZERO')
```

will perform the connection of file RESULTS to unit 2000. The connection will be defined for sequential formatted input/output with any spaces in numeric fields being ignored apart from such fields which consist entirely of spaces. The effect of the second OPEN statement is to change the interpretation of the spaces. No other property of the connection is affected.

Example 3

The sequence

```
OPEN(10,FILE='OUTPUT',...)
⋮
OPEN(44,FILE='OUTPUT',...)
```

is not permitted as it attempts the simultaneous connection of the file OUTPUT to two units.

16.8.3 The CLOSE Statement

The CLOSE statement terminates the connection of a unit, and hence terminates the connection of the file to which it is connected. The statement has an option to destroy the associated file, that is, to cause it not to exist after the statement has been executed. Note that once a file has been disconnected it will be free to be connected again (provided that it still exists), either to the same unit or to another unit.

The general form of the CLOSE statement is:

```
CLOSE(UNIT=k,IOSTAT=ios,ERR=errs,STATUS=sta)
```

where:

UNIT=*k* is an integer expression that identifies the unit to be disconnected. Its value must be zero or positive. The characters **UNIT=** may be omitted but in this case the

unit specifier must be the first item in the list, otherwise the position of the specifier is not fixed.

The remaining specifiers are optional, and if they are defined they may appear anywhere in the list. The specifiers are described below together with any assumed value or action that may be taken if a specifier is not defined:

IOSTAT=*ios* is an input/output status specifier that defines an integer variable or integer array element which becomes assigned with a positive or zero value. When an error condition exists the input/output status specifier is assigned a positive value which corresponds to an error message which describes the error. When no error exists it is set to zero.

ERR=*errs* defines a statement label to which control is transferred if an error condition is detected. If both the error specifier and the input/output status specifier are omitted then program execution will terminate when an error occurs.

STATUS=*sta* is a status specifier. *sta* is a character expression whose value may be either KEEP or DELETE; any trailing spaces are ignored. If DELETE is specified the connected file will cease to exist; DELETE is the only value that may be specified for a file which is a temporary file, that is one whose status was SCRATCH before the CLOSE statement. If KEEP is specified for a preconnected file which does not exist, the file will still not exist after the CLOSE; otherwise if it is specified for an existing file then the file will continue to exist.

If the status specifier is omitted the assumed value is KEEP, unless the file status prior to the CLOSE statement was SCRATCH in which case the assumed value is DELETE.

It is quite permissible in a CLOSE statement to specify a unit which is not connected. It has no effect on the unit and it affects no file.

When program execution terminates, each unit that is still connected is closed with STATUS=KEEP, unless the file to which it is connected is a temporary file, that is one which was created with STATUS=SCRATCH, in which case it is closed with STATUS=DELETE. Note that the effect is the same as if the unit was specified in a CLOSE statement with no status specifier defined.

16.8.4 The INQUIRE Statement

The INQUIRE statement is used to interrogate the properties of a particular file or the properties of the connection to a particular unit. The given file or unit need not be connected. There are two forms of the INQUIRE statement:

- INQUIRE by unit
- INQUIRE by file

An INQUIRE by unit statement interrogates a specified unit and the file to which it is connected if any; while the INQUIRE by file statement interrogates the properties of a specified file which may or may not exist.

16.8.4.1 INQUIRE by Unit

The general form of an INQUIRE by unit statement is:

```
INQUIRE(UNIT=k,slist)
```

where:

UNIT=*k* is the unit specifier which defines an integer expression. The value of the integer expression must be either zero or positive and specifies the unit number being

inspected. The characters **UNIT=** may be omitted in which case the unit specifier must occur in the position indicated above; otherwise it may appear anywhere within *slist*.

slist is a set of **INQUIRE** specifiers (see section 16.8.4.3) which may be an empty list. Each specifier which is not omitted inquires about a particular property of the specified unit or file to which it is connected.

16.8.4.2 INQUIRE by File

The general form of an **INQUIRE** by file statement is:

```
INQUIRE(FILE=fn,slist)
```

where:

FILE=*fn* specifies the name of the file being interrogated and may appear anywhere in *slist*. *fn* is a character expression whose value represents the file name; any trailing spaces in the value are ignored. The file may or may not either exist or be connected.

slist is a list of **INQUIRE** specifiers (see section 16.8.4.3) which may be empty. Those specifiers which are defined inquire about a particular property of the file.

Note that because *fn* is a character expression, the character '\' is interpreted as an escape character, as described in section 9.2.1.11. This means that if you wish to use an MS-DOS pathname when specifying the file to open, you must double the '\' character. For example:

```
INQUIRE (FILE='C:\\DATA\\SET12.DAT', EXIST=EXI)
```

16.8.4.3 The INQUIRE specifiers

Any of the specifiers defined below may be used with either form of the **INQUIRE** statement. Each specifier is optional, and may appear anywhere within the list of **INQUIRE** specifiers. A description of the specifiers follows.

IOSTAT=*ios* where *ios* is an integer variable or array element which becomes defined with a value of zero if no error condition exists or with a positive value if an error condition was detected. *ios* is known as an input/output status specifier. If it becomes defined with a positive value then the value identifies an error message which describes the error encountered.

ERR=*errs* where *errs* is an error specifier that defines a statement label to which control is transferred if an error condition exists. If no error specifier or input/output status specifier is defined then program execution will terminate when an error occurs.

EXIST=*ex* where *ex* is a logical variable or logical array element. For an **INQUIRE** by file statement it is assigned the value **.TRUE.** if the specified file exists, otherwise it is assigned the value **.FALSE.** If the inquiry relates to a unit, *ex* is always assigned the value **.TRUE.**, that is, any unit number that is zero or positive may be used for input/output. Note that some implementations may provide a more restricted set of unit numbers.

OPENED=*op* where *op* is a logical variable or logical array element that is always assigned a value when this specifier is defined. In an **INQUIRE** by file statement it is assigned the value **.TRUE.** if the specified file is connected to a unit and the value **.FALSE.** if it is not connected to any unit. Similarly, in an **INQUIRE** by unit statement it is assigned the value **.TRUE.** if the specified unit is

connected to a file, and the value .FALSE. if it is not currently connected.

NUMBER=*num*

where *num* is an integer variable or integer array element which only becomes defined with a value if the specified unit or file is currently connected, in which case it is assigned with the number of the connected unit. If there is no connection *num* becomes undefined.

NAMED=*nmd* where *nmd* is a logical variable or logical array element. For an INQUIRE by unit statement *nmd* becomes undefined only if the unit is not connected to a file, while for an INQUIRE by file statement it becomes undefined only if the specified file does not exist. When *nmd* does become defined, it is assigned the value .TRUE. if the file exists and has a name; otherwise it is assigned the value .FALSE.

NAME=*nm* where *nm* is a character variable or character array element which is assigned the name of the file only if the NAMED= specifier may be assigned the value .TRUE.; otherwise *nm* becomes undefined. Note that the value assigned to *nm* may not necessarily be the same as that defined by the FILE= specifier in an INQUIRE by file statement; however, it will always represent an acceptable value for the file identifier in an OPEN statement.

ACCESS=*acc* where *acc* is a character variable or character array element which only becomes defined if a connection exists. *acc* is assigned the value SEQUENTIAL if the connection is for sequential input/output, or it is assigned the value DIRECT if the connection is for direct access input/output.

SEQUENTIAL=*seq*

where *seq* is a character variable or character array

element. It is assigned the value YES if sequential access is one of the permitted forms of access method for the file. A value NO is assigned if the file may not be accessed sequentially. If the access property of a file cannot be determined, a value of UNKNOWN is assigned.

In Parallel Fortran the sequential specifier is generally assigned the value YES, that is any file may be opened for sequential access. However in certain circumstances the value UNKNOWN is returned if it is not known whether the file is suitable for sequential access.

DIRECT=*dir* where *dir* is a character variable or character array element which is assigned the value YES if direct access is one of the permitted forms of access method for the file. A value of NO is returned if direct access is not one of the properties of the file. If for some reason the access property of the file cannot be determined the value UNKNOWN is assigned.

In Parallel Fortran, *dir* is assigned the value YES if the file is currently connected for direct access; otherwise, in general it is assigned the value UNKNOWN as it is not known whether the file is suitable for direct access.

FORM=*fm* where *fm* is a character variable or character array element which describes the form of the current connection. It is assigned the value FORMATTED if the connection is for formatted input/output statements, and the value UNFORMATTED if the connection is for unformatted input/output statements. *fm* becomes undefined if there is no connection.

FORMATTED=*fmt*
where *fmt* is a character variable or character array element. It describes whether formatted input/output statements may be used to access the file. If the file

consists of formatted records, *fmt* is assigned the value YES, while if the file consists of unformatted records it is assigned the value NO. In some circumstances it may not be possible to determine the permitted form and *fmt* will be assigned the value UNKNOWN.

If the file is currently connected for formatted input/output Parallel Fortran returns the value YES. Otherwise the value UNKNOWN is returned as it is not known whether the file is suitable for formatted input/output.

UNFORMATTED=*unf*

where *unf* is a character variable or character array element that is assigned the value YES if unformatted input/output statements may be used to access the file. Otherwise it is assigned the value NO when only formatted input/output statements can be used. In circumstances when the permitted form cannot be determined *unf* decomes defined with the value UNKNOWN.

In Parallel Fortran the value YES is returned if the file is currently connected for unformatted input/output. In all other circumstances it returns the value UNKNOWN as it is not known whether the file is suitable for unformatted input/output.

RECL=*rl* where *rl* is an integer variable or integer array element which becomes defined only if there is a connection and the connection is for direct access, otherwise it becomes undefined. *rl* is assigned the value of the record length of the file in units of characters (bytes).

NEXTREC=*nr* where *nr* is an integer variable or integer array element. If the file is connected and the connection is for direct access, *nr* is assigned the record identifier of the record which follows the last record that was written or read. Otherwise *nr* becomes undefined. Note that if no

record has been accessed since the file was connected a value of 1 is assigned.

BLANK=*blk* where *blk* is a character variable or character array element and applies only to a connection which is for formatted input/output. If any spaces within a numeric input field are to be ignored than *blk* is assigned the value **NULL**. If spaces other than leading spaces are to be interpreted as zeroes than *blk* is assigned the value **ZERO**. *blk* becomes undefined if there is no connection or if the connection is for unformatted input/output.

When using an **INQUIRE** by unit statement or an **INQUIRE** by file statement, the following points should be noted:

- Unless an error condition exists, the opened specifier (**OPENED=**) and the exist specifier (**EXIST=**) always become defined. If an error condition does exist then all the **INQUIRE** specifier values become undefined.

- No variable or array element may be defined more than once in the list of **INQUIRE** specifiers.

- If the exist specifier and the opened specifier become defined with the value **.TRUE.** within an **INQUIRE** by unit statement, then all the other specifiers become defined.

- Within an **INQUIRE** by file statement, if the opened specifier is assigned the value **.FALSE.** (that is, if the file is not connected), then the number, access, form, record length, next record and blank specifiers become undefined. Note that if the file is connected the record length, next record and blank specifiers need not become defined.

- Within an **INQUIRE** by file statement, if the exists specifier is assigned the value **.TRUE.**, then the named name, sequential, direct, formatted and unformatted specifiers become defined; otherwise they become undefined.

Example

Assume that the statement

```
OPEN(FILE='DATAFILE',ACCESS='DIRECT',RECL=80,UNIT=730)
```

has been successfully executed and has connected an existing file DATAFILE to unit 730. The properties of the connection are defined as direct access input/output with unformatted records whose length is 80 bytes. Also assume that no other connection or preconnection exists.

Then

```
INQUIRE(UNIT=9,EXIST=L1,OPENED=L2,DIRECT=C1)
```

will assign the value .FALSE. to L2 as unit 9 is not connected and consequently C1 becomes undefined. L1 will be assigned the value .TRUE. as the specified unit may be used for input/output.

```
INQUIRE(FILE='INPUTS',OPENED=L1,NUMBER=I1)
```

will assign the value .FALSE. to L1 as the file INPUTS is not connected to a unit and I1 will therefore become undefined. Note that the file's existence has no effect on the values set.

```
      INQUIRE(BLANKS=C1,NEXTREC=I1,ACCESS=C2,FORMATTED=C3,
     1        FILE='DATAFILE',OPENED=L1,EXIST=L2)
```

interrogates the connection defined above and consequently L1 and L2 are assigned the value .TRUE., and C2 the value DIRECT. C3 is assigned the value NO which causes C1, the blank specifier, to become undefined as the connection is for unformatted input/output. The value assigned to I1 defines the current position of the file. If no records have been accessed since the connection was defined I1 is assigned the value 1; otherwise, it is assigned the record identifier of the record which follows sequentially after the last record accessed.

Part IV

General Reference

Chapter 17

Fortran Compiler Reference

This chapter contains technical information about the way the Fortran compiler works. Full details of the language itself may be found in part III. Note that the information in this chapter applies only to the current version of the compiler; it is not guaranteed that future versions of the compiler will behave in the same way.

17.1 Running the Compiler

The compiler is run by either of the commands `t4f` or `t8f`.

`t4f` generates object code for the T414 32-bit transputer.

`t8f` generates object code for the T800 floating-point transputer.

The command line used to invoke the compiler must specify a single source file name. Wild cards are not allowed.

Option switches may optionally be given on the command line. Option switches are introduced by the '/' character; the available switches are discussed in section 17.2 below.

If the source file is successfully compiled, a zero exit status code is returned to DOS. If errors are detected, the compiler returns an exit status code of 1. This feature can be used in DOS batch files to check whether a compilation was successful.

The compiler creates a number of temporary files as it works. Normally, these are placed in the current directory; however, the environmental variable **TMP** may be used to make the compiler put them in another directory. For example, to make the compiler place the temporary files in the root directory on disk `D:`, the following DOS command could be used.

```
C>set TMP=D:\
```

The temporary files are called `ftemp.1`, `ftemp.2` and `ftemp.3`. Usually, the compiler will delete them at the end of the run, but occasionally this may not be done; in this case, it is safe to delete them yourself.

17.2 Compiler Switches

When the compiler is invoked, the user may include in the MS-DOS command one or more *switches* to control its behaviour. This section describes these switches.

Following the MS-DOS convention, switches are introduced by a '/' character and may be typed in any order, before or after the source file specification. Switches and their argument strings are not case-sensitive; that is, lower-case letters have the same significance as the corresponding upper-case letters. This means, for example, that the following two switches would be treated the same:

```
/FBhello.bin
/fbHELLO.BIN
```

The format of the various switches is described using the following notations:

fn	An MS-DOS filename. It may be omitted in whole or in part; the compiler's behaviour in this case is described in section 17.2.3 below.
dir	An MS-DOS filename, which will be assumed to refer to a directory.
n	An integer. By default this is decimal, but hexadecimal numbers may be input, using the notation **16_*n***.

An example of a command to invoke the compiler with switches:

```
C>t4f hello /q /flKEEP /lx
```

This will invoke the T4 compiler to compile **hello.f77**, and place a listing of the source file with any error messages and a cross-referenced symbol table in **keep.lis**. Warning messages on the console are suppressed.

17.2.1 Default switches

Switches are normally entered on the command line when the compiler is invoked. In practice, you may find you use some switches on every compilation. To avoid entering the same switches again and again, the compiler also allows switches to be entered through a DOS environmental variable. The contents of the environmental variable **TF**, if any, are prefixed to the arguments supplied on the command line. For example, to make the compiler print its version number (**/I**) and generate debug tables (**/Zi**) every time it is run, give DOS the command: **set TF=/i/Zi**

Default options set in this way can be turned off again using the DOS command: **set TF=**

17.2.2 Controlling Source Processing

17.2.2.1 Switch /R

If this switch is used, the compiler allows source lines which are up to 132 characters wide, rather than 72. See section 8.3.1.

17.2.2.2 Switch /D

If this switch is used, debug comment lines are compiled as normal source lines. A debug comment line has a 'D' in column 1. For further information, see section 8.3.1.3.

17.2.2.3 Switch /U

The effect of this switch is to stop the predefined specification of the types of symbolic constants, variables, arrays and functions. (For a discussion of predefined specification, see section 9.3.1).

If /U is not specified, things whose names start with I, J, K, L, M and N are predefined as being of type integer; all other things are predefined as being of type real. If /U is specified, this predefined specification is not done, and all names must be defined with the IMPLICIT statement or one of the explicit type statements; any name which is not defined in this way is flagged by the compiler with error 333, *name* `must be explicitly typed`.

17.2.3 Controlling Output Files

The /F (File) switch is used for specifying which output files are to be generated, and their names. Each of the varieties of /F may be followed by a *fn*, but the complete MS-DOS path name may not be necessary. The compiler supplies defaults, as follows:

- If no extension is given, the compiler supplies a default extension depending on the type of output file: `.lis` for listing files, etc.

- If no filename is given, the filename of the source file is used.

- If no drive specification and no directory specification are given, the drive and directory specification of the source file are used; if the source file specification did not include a drive and/or a directory specification, then the current drive and/or directory are used.

- If a drive specification is given alone, then the output file is created in the current directory of the specified drive, regardless of the source file's directory.

The following examples may clarify this. The 'Supplied' string below is assumed to be the argument of a **/FL** switch. The current drive and directory are `c:\michael`, and the current directory on `a:` is `\output`.

Specified source file	Supplied	Output file
`dogs`	*nothing*	`c:\michael\dogs.lis`
`dogs`	`cats`	`c:\michael\cats.lis`
`dogs`	`cats.out`	`c:\michael\cats.out`
`dogs`	`\stuff\`	`c:\stuff\dogs.lis`
`dogs`	`\stuff`	`c:\stuff.lis`
`dogs`	`a:\first\`	`a:\first\dogs.lis`
`dogs`	`a:`	`a:\output\dogs.lis`
`dogs`	`a:cats`	`a:\output\cats.lis`

Notice that in examples like the fourth above, it is the fact that the supplied string ends with a '\' which indicates that this is a directory specification. If it is omitted, as in the fifth example, output would be sent (in this case) to `c:\stuff.lis`, even if a directory `c:\stuff` exists.

17.2.3.1 Switches /FB and /FO

These switches have the same effect. They instruct the compiler to create an object file in binary format. The default extension is `.bin`.

Notice that if no **/FB** or **/FO** switches are specified, the behaviour of the compiler is the same as if a **/FB** switch were used, with no argument. In order to stop the compiler generating an object file of any kind, the **/C** switch must be used (see section 17.2.4).

17.2.3.2 Switch /FH

This switch makes the compiler produce an object file in the legible hexadecimal format described in section 3 of the Inmos Standalone Compiler Implementation Manual. It may not be specified with the **/FB** or **/FO** switch. The default extension is **.hex**.

Hexadecimal object files can be linked with normal binary object files as described in section 3.3.

Normally, the **/FH** switch is only used if it is necessary to transmit an object file across a communication channel which does not support transparent binary file transfer. A hexadecimal object file is much larger than the corresponding binary object file.

17.2.3.3 Switch /FL

This switch makes the compiler produce a line-numbered source listing file. The listing file contains any error messages produced by the compiler, as well as the numbered source lines. The default extension is **.lis**.

For example, if the `hello.f77` program were compiled using this command:

```
C>t4f hello/fl
```

the listing file produced would look like this:

```
Source file: hello.f77
Object file: hello.bin
Switches:    /T4 /FL
Compiled by: transputer Fortran 77 compiler, F77_transputer V2.1

     1      C 'Hello, world' program
     2      C
     3            PRINT 100
     4      100   FORMAT ('Hello, world!')
     5            END
     6
```

17.2.4 Controlling Object Code

17.2.4.1 Switches /T4, /T8 and /T8A

These switches can be used to specify which type of transputer the program is to be compiled for. **/T4** and **/T8** are only permitted with the **tf** command, as the **t4f** and **t8f** commands supply the appropriate switches automatically, and these will, in fact, appear in the **Switches:** line of the listing (see section 17.2.3.3 above).

The **/T8A** switch is valid with the **t8f** and **tf** commands. It makes the compiler generate code to work round a floating-point firmware bug in Rev A of the T800 processor which affects integer-to-real conversions.

17.2.4.2 Switch /S

The **/S** (Save) switch makes the compiler allocate all scalar variables (that is, variables which are not arrays) to static storage, so that they keep their values between calls to a subprogram. Variables in the following categories are always allocated to static storage in any case:

- arrays;
- variables initialised by **DATA** statements, or by the extended forms of the explicit type specification statements (see section 10.3.1.8);
- variables in **COMMON** blocks;
- variables which have been the subject of **SAVE** statements.

If the program is compiled without the **/S** switch, all other variables are allocated to stack storage, and so are undefined on entry to a subprogram, until they have been assigned a value, for example by an assignment statement. Many Fortran compilers in fact preserve the values of all variables across calls, although the standard does not guarantee this. The **/S** switch provides compatibility with compilers of this sort.

A similar problem arises from the fact that some compilers set all uninitialised variables to zero, although the standard does not require this. With Parallel Fortran, uninitialised static variables are preset to zero. If a ported program is failing because variables are not being preset to zero, you can compile the program with the **/S** switch. This will allocate all variables to static, so that they are all preset to zero.

Unfortunately, accessing static storage is slower than accessing the stack, and for this reason it may be preferable to use a **SAVE** statement (see section 14.6) to preserve those variables which need preserving, rather than to compile with the **/S** switch. On the other hand, it is sometimes important to minimise the amount of stack space used (see section 3.4), in which case this switch may be what is needed.

17.2.4.3 Switch /P

Certain constant values in a program cannot be worked out by the compiler, but must be filled in (or *patched*) by the linker. The compiler leaves gaps for these values, and fills the gaps with a special

code. In some circumstances, however, the linker may decide on a patch value which is too large to fit in the gap provided by the compiler. When this happens, the linker gives the following error message:

FATAL ERROR(22): patch over valid code in module *module*

The **/P** switch controls the sizes of the gaps left by the compiler, so that this situation can be avoided. There are two varieties.

17.2.4.4 Switch /PC*n*

This switch changes the size of the gap the compiler leaves for a call to a subprogram. The size of the gap limits the distance from the call to the called subprogram. Four bits of the *displacement* are stored in every byte of gap, so the maximum displacement is $2^{4n} - 1$ bytes. n should be in the range 2 to 8. If the **/PC** switch is not used, the compiler assumes a value of 6 for n, giving a maximum displacement of 16MB. Similar negative displacements are also allowed. Smaller values of n reduce the code size for external calls (resulting in faster execution) but restrict the total size of the final program image. For example, $n = 5$ allows displacements up to 1MB; $n = 4$ allows up to 64KB. Normally the default value of n should be adequate.

The compiler does not accept a **/PC1** switch, as in this case not only would the displacement be restricted to 15 bytes, but in addition backward calls would not be possible.

17.2.4.5 Switch /PM*n*

A linked program contains a *module table*, which has an entry for every module in the program, including both the modules written by the user and those extracted from libraries. Each module's entry contains the address of the module's static data area. The first thing which a subprogram does is to access this address, and to do this, it must load the *module number*. These module numbers are assigned

by the linker, so the compiler cannot predict how large a module's number will be. Once again, it leaves a gap, and the **\PM** switch allows the user to specify how large this gap is. Four bits of the module number are stored in every byte of gap, so the maximum module number is $2^{4n} - 1$ bytes. n should be in the range 2 to 8. If the **/PM** switch is not used, the compiler assumes a value of 2 for n, giving a maximum module number of 255. Larger **/PM** numbers increase the maximum number of modules which can be linked into one program, but make the program slightly larger and slower.

If the linker reports **patch over valid code**, as described above, the likely cause is that the linked program contains more than 255 modules, including library modules. The programmer can cope with this situation as follows:

- Use **/PM** to increase the maximum allowable module number. For example, **/PM3** will allow 4096 modules.

- Modules are assigned numbers in order, depending on their position in the linker's command line. It is essential that modules from the Fortran library should have module numbers which are less than 255; they have already been compiled with **/PM2**, and this cannot be changed. So the linker command line should have the Fortran library and harness first; then any user-written modules and libraries, compiled with a larger **/PM**. For example:

```
C>linkt \tf2v1\frtlt8 \tf2v1\t8harn main @mysubs,main.b4
```

17.2.4.6 Switch /C

If the **/C** (Check) switch is used, the compiler checks the source file for errors, but does not generate an object file.

17.2.5 Controlling Debugging

17.2.5.1 Switch /Zi

If this switch is used, the object file generated by the compiler will contain tables listing the names, locations and types of the identifiers used in the source file. This switch should be used if Tbug, 3L's interactive symbolic debugger is to be used to debug the compiled code. By default, no tables are generated and the debugger is unable to display source program variables symbolically.

17.2.5.2 Switch /Zd

This switch causes line number tables relating compiled code addresses back to source line numbers to be included in the generated object file for use by **decode** and Tbug.

In this release, line number tables are always included in the generated object file. However, as future releases may not generate line number tables by default, you may find it helpful to get into the habit of using **/Zd** now.

17.2.6 Controlling INCLUDE Processing

This section should be read in conjunction with section 17.3, where include file processing is discussed more fully. The INCLUDE statement is discussed in section 14.7.

17.2.6.1 Switch /I*dir*

This switch adds *dir* to the include list, that is, the list of "standard places" where the compiler looks for files specified in INCLUDE lines. The *dir* string is assumed to be a directory, whether or not it terminates with a '\'.

17.2.6.2 Switch /X

This switch excludes the "standard places" from the include list. Directories added to the include list by means of the /I*dir* switch are not affected, and will still be searched by the compiler.

17.2.7 Controlling the Format of the Listing

The /L switch can be used to specify changes in the information to be included in the listing file output by the compiler. If /L is used without /FL being used as well, the compiler behaves as if /FL were specified without a filename; that is, the listing is sent to a file whose name is derived from the source file name (see section 17.2.3).

Note that combinations of these two switches (/LIX or /LXI) are permitted.

17.2.7.1 Switch /LI

Normally, the contents of files which have been added to the source program by INCLUDE statements (see section 14.7) are not included in the listing file. When /LI (List Include) switch is used, however, they are included.

17.2.7.2 Switch /LX

The /LX (List Cross-References) switch causes the compiler to generate an alphabetical list of all identifiers, specifying their types (see section 9.1) and the lines on which they are referenced.

17.2.8 Information from the Compiler

17.2.8.1 Switch /I

This switch makes the compiler display a line containing its identity and version. Please quote this information in any correspondence about the compiler.

17.2.8.2 Switches /V and /Q

These switches control the information sent to the standard output stream by the compiler during compilation. If neither **/V** nor **/Q** is used, error messages only, including warning-class and comment-class errors (see 17.6) are output.

The **/V** (Verbose) switch makes the compiler produce additional messages on the standard output stream indicating how far compilation has progressed. The compiler also outputs the name of each subprogram as it is compiled. Here is a typical example of output generated by the **/V** switch:

```
    program    MAINPROG
    subroutine SECOND
    function   THIRD
 17 statements analysed; no faults detected
starting object file generation
```

The **/Q** (Quiet) switch stops the compiler from sending warning-class and comment-class error messages (see 17.6) to the standard output stream. Error-class messages are still output, and all messages are still output to the listing file, if any.

17.2.9 Controlling the Compiler's Buffer Sizes

The **/B** (Buffers) switch enables the user to change the size of some of the compiler's internal buffers. The default sizes for these buffers

will be appropriate for the majority of programs, but if a program which includes particularly large or complex subprograms is compiled without the use of the /B switch, the compiler may be unable to compile the program. These conditions may be recognised by the occurrence of one of the following error messages:

1. **Dictionary table overflow**
2. **Triad table overflow**
3. **Fatal error -- too many code fragments**

The format of the /B switch is as follows:

```
/B(a:b:c)
```

where:

a	is the requested size of the dictionary table;
b	is the requested size of the triad table;
c	is the requested size of the code fragment buffer.

All these sizes are expressed in kilobytes. If any value is omitted, the default value is left unchanged. Trailing ':' characters after the last value may be omitted. For example, the following switch would specify the size of the triad table as 100 kilobytes, and leave the other two buffers unchanged:

```
/B(:100)
```

The characteristics of the three buffers are defined by the following table.

	Minimum Size	Maximum Size	Default Size
Dictionary table	32	128	64
Triad table	60	240	120
Code fragment buffer	16	*no limit*	16

Note that it is possible to specify combinations of the three buffers which will overflow the transputer's memory. If this happens, the following error message will be given:

```
Fatal Error -- not enough memory
```

The /B switch may also be used alone, without parameters and parentheses; in this case, a summary of its format and the table above is printed.

17.2.10 Obsolescent Switches

Two switches are provided for compatibility with earlier versions of the compiler.

/H This is the equivalent of /FH. Like /FH, it cannot be used with /FB.

/L Similarly, this is the equivalent of /FL.

Neither of these two switches can have any arguments, and consequently they cannot be used to redirect output to a hexadecimal or listing file.

17.3 Handling of INCLUDE Files

This section discusses how the compiler's handling of INCLUDE statements (see 14.7) may be controlled.

When the compiler encounters an INCLUDE statement, it searches for the specified file in a sequence of directories known as the *include list*. This consists of the following, which are searched in this order:

1. The current directory.
2. The "standard places". These are defined in one of three ways.

- The user may define the string **3LF_INC** in the MS-DOS environment to specify a series of directories, by an MS-DOS command like this:

```
C>SET 3LF_INC=c:\root\branch;\cats
```

- If **3LF_INC** is not defined, there is only one standard place: directory **\tf2v1**. If **3LF_INC** has been defined, **\tf2v1** is only searched if it is included in the series of directories it specifies.
- If the **/X** compiler switch is used, the standard places are excluded from the include list.

3. Directories which have been specifically added to the include list at compilation time by means of the **/I** switch (see section 17.2.6).

All the filenames which are added to the include list, either by the **SET 3LF_INC=** command or by the **/I** compiler switch are assumed to be directories, even if they do not end with a '\'; in this case, the '\' is supplied by the compiler. If the filename specified in the **INCLUDE** statement includes a directory specification, an attempt is made to concatenate it to each of the directories in the include list in order to find the file. Such a filename should not itself start with a '\'.

After installing the compiler as described in chapter 1, only the directory **\tf2v1** will be searched. Note that this directory specification, which is built into the compiler, does not include a disk name: this implies that the compiler will only search directory **\tf2v1** on the *current* disk. If the compiler is to be executed from a different disk to that on which it has been installed, the string **3LF_INC** must be defined to allow the compiler to locate the standard package files (see section 18.2.1).

Because the filename in an **INCLUDE** statement is a Fortran character constant, the character '\' is interpreted as an escape character, as described in section 9.2.1.11. This means that if you wish to use

an MS-DOS pathname when specifying the included file, you must double the '\' character. For example:

```
INCLUDE 'c:\\lib\\incfiles\\maths.inc'
```

17.4 Data-Type Representations

This section briefly expands on the discussion of Fortran data types to be found in section 10.1. The Fortran data types are represented on the transputer as follows:

`BYTE`	1 byte	
`INTEGER`	1 word	
`REAL`	1 word	(IEEE single-precision)
`LOGICAL`	1 word	
`DOUBLE PRECISION`	2 words	(IEEE double-precision)
`COMPLEX`	2 words	(2 IEEE single-precisions)
`DOUBLE COMPLEX`	4 words	(2 IEEE double-precisions)
`CHARACTER*`*n*	*n* bytes	

On T4 and T8 transputers, a byte is 8 bits and a word is 32 bits. Variables of all types are automatically word-aligned, except as noted below.

The IEEE floating-point formats used to hold `REAL`, `DOUBLE PRECISION`, `COMPLEX` and `DOUBLE COMPLEX` quantities on the transputer are described in detail in the IEEE floating-point standard[9]. The way in which these standard formats are represented in transputer memory is shown in figure 17.1.

In the case of `COMPLEX` quantities, two IEEE single-precision quantities are used; the real part is stored in the word with relative address displacement 0, and the imaginary part is stored in the word with relative displacement 4. Similarly, `DOUBLE COMPLEX` quantities are held as two IEEE double-precision quantities, with the real part at relative address displacement 0 and the imaginary part at relative address displacement 8.

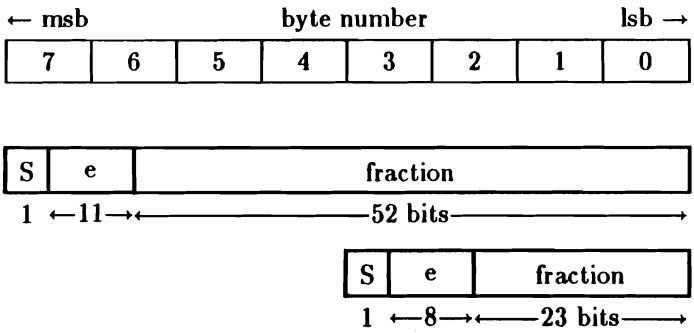


Figure 17.1: Representation of **DOUBLE PRECISION** and **REAL** Values

LOGICAL variables each occupy a word. The value **.TRUE.** is represented by a value 1 in the least-significant bit, and the value **.FALSE.** is represented by a 0. All other bits have the value 0.

CHARACTER variables are represented by a string of bytes whose length is the length of the variable, rounded up to the nearest 8 bytes.

Arrays are stored as described in section 10.1.2, that is, in ascending address order, with the lefthand subscript varying most rapidly and successive subscripts varying less rapidly. Each element of the array is stored as described above, according to its type, except that:

- **CHARACTER** elements do not have their lengths rounded up to the nearest 8 bytes;
- **BYTE** elements are not aligned on word boundaries, but follow one another without any gaps.

17.5 Data File Formats

This section describes how the various file types are realised in the MS-DOS file system.

17.5.1 FORMATTED SEQUENTIAL

Each record consists of a sequence of ASCII characters, terminated by a carriage return-line feed sequence. Files generated by this version of Parallel Fortran terminate the file with a Ctrl/Z character.

17.5.2 FORMATTED DIRECT

The record consists of a sequence of ASCII characters, with no terminating carriage return-line feed. If the record is not filled by the character sequence, the remaining bytes are filled with space characters. There is no end-of-file marker. If a record is written beyond the current highest-numbered record, the contents of any intervening records are undefined.

17.5.3 UNFORMATTED SEQUENTIAL

The variables written to a record are stored in the format they have in memory. However, they are not word-aligned, and **CHARACTER** values are not rounded out to a multiple of 8 bytes. Each record is preceded and followed by a 4-byte integer record length; this record length does not include the lengths of the record-length words themselves. Files generated by this version of Fortran have an end-of-file marker: this is four bytes containing the value -1.

17.5.4 UNFORMATTED DIRECT

Variables are stored in the records in the same way as for **UNFORMATTED SEQUENTIAL** files, but there are no record length indicators. If the record is not filled with data, the remaining bytes are filled with zeroes. There is no end-of-file marker. If a record is written beyond the current highest-numbered record, the contents of any intervening records are undefined.

17.6 Fortran Error Messages

This section is concerned with the format of error messages output by the Parallel Fortran compiler at compilation time, and by compiled programs at run time. These messages fall into four main groups, which will be discussed in turn:

Syntax Errors These messages report errors or problems in the syntax of the program, and occur during the first part of the compilation process, when the program is being analysed syntactically.

Code Generation Errors These occur when the compiler encounters a problem while generating the object file.

Fatal Errors These errors result in the immediate termination of the compilation, and mostly originate in the interaction between the compiler and the operating system.

Run-Time Errors These occur when a Fortran program is run, and result either from programming errors, or from interactions between the program and the operating system at run time.

17.6.1 Syntax Errors

There are three classes of syntax error messages output by the compiler. By default, they are all output to the console, and to the listing file, if any. However, Warning- and Comment-class messages are not output if the `/Q` switch is used.

Errors The compiler output a message of this class when it attempts to compile a statement which does not obey all the rules of Fortran. The message identifies the error and attempts to indicate where it took place. After it has output the message, the compiler continues to analyse the program, but no object file will be produced.

Warnings A message of this class is used by the compiler to identify a statement which does not conform to strict ANSI standard, although it is acceptable to the Parallel Fortran compiler. After the message, the compiler continues to analyse the program, and will produce an object file.

Comments A comment message is used by the compiler to draw the attention to an effect which the programmer may not have intended, and which may be the result of a programming error.

Syntax error messages have the following format:

ln ... *source-text*...
^

class *errno* ... *message-text*...

where:

ln is the number of the source line where the error happened.

source-text is the text of the erroneous line.

class is one of the words **Error**, **Warning** or **Comment**.

errno is a number identifying the error.

message-text is the message, describing the nature of the error.

The '^' points to the part of the source text which is in error. For example:

```
2           A(1.5) = 10
                ^
Error 131  Expression must be of type integer
```

In this case, the programmer has attempted to subscript an array with an expression which is not of type integer. This is not allowed by the rules of Fortran (see section 9.2.4). The error occurred on line 2 of the source file, and was noticed by the compiler when it had

finished evaluating the expression, that is, when it encountered the closing ')' of the subscript. This particular error is number 131, and the message will be found under this number in appendix G.

Appendix G contains a complete list of syntax error messages, arranged in *errno* order.

17.6.2 Code Generator Errors

Once the syntactic and semantic phases of compilation have been completed the compiler attempts to generate transputer instructions for the program.

During this phase of compilation the compiler does not have access to the source program and so error messages cannot include the offending statement but simply give its line number.

The general form of these messages is:

Error: ... *reason* ... **at line** *number*

The current version of the compiler can output only one code generation message. It is treated as a fatal error; that is, compilation stops at this point.

- **Zero divide**

  This message is issued when the compiler tries to evaluate a constant expression and discovers that the divisor is zero.

```
PROGRAM WRONG
INTEGER A
A = 5 / 0
END
```

17.6.3 Fatal Errors

Certain error conditions are so serious that the compiler cannot carry on. Instead, it gives one of the error messages listed in this section,

and stops. The errors are associated with the compilation process itself and are independent of the Fortran language.

Most of the messages are introduced by the phrase:

```
Fatal Error --
```

In the list below, those which do not include this introduction are marked with a †.

- **cannot produce both hex AND binary files**

  The user included on the compiler's command line switches which requested both hex and binary format object files. For example:

  ```
  C> t4f mangle /fo /fh
  ```

- **† Dictionary table overflow**

  The dictionary table is used by the part of the compiler which performs syntactic analysis to store details of the various Fortran entities it is dealing with. If this message occurs, the size of the dictionary table can be increased using the **/B** switch (see section 17.2.9).

- **error in format of /B switch:** *switch*

  Section 17.2.9 describes the format of the **/B** switch.

- **file system error:** *n*

  This error occurs when the **afserver** reports to the run-time library that a failure in the MS-DOS file system has happened. *n* is the error code returned by the **afserver**; a list of error codes and their meanings may be found in the *Standalone Compiler Implementation Manual*[13]. The most common cause of this error message is an attempt to write to a full disk.

- **more than one source file specified**

  Only one source file can be specified for each compilation. Notice that any item on the compiler's command line which does not start with a '/' is treated as a source file, so a common cause of this error is inserting spaces where they are not expected, like this:

  ```
  C> t4f mangle /FO mangle.out
  ```

  In this case, the user presumably meant to type this:

  ```
  C> t4f mangle /FOmangle.out
  ```

- † **Name table overflow**

  The name table is the area used by the syntactic analyser to store the text of the names of the Fortran entities it is handling. The table is very large, and consequently this message is expected to occur only in extreme examples. The only cure is to recast the program so that the subprogram which is causing the problem uses fewer (or shorter) names. The offending subprogram can be identified by using the **/V** switch.

- **not enough memory** This message indicates that the compiler and all its data areas are too large for the B004's memory. The most likely cause is that one or more of the buffers adjustable by the **/B** switch has been set too large. See the description of the **/B** switch in section 17.2.9.

- **range for patch size is 2 to 8 bytes**

  The message indicates that the **/PC***n* compilation switch has been specified with an invalid value for the displacement value *n*. Refer to section 17.2.4.4 for a discussion of this switch.

- **switch should contain a number:***switch*

The **/PC** and **/B** switches have numeric parameters. This message indicates that this number is badly formatted.

- **target must be /T4 or /T8 only:***switch*

  This message is output when a switch starting with a 'T' is not recognised as a valid target identifier. For example:

  ```
  C> tf mangle /TOAD
  ```

- **target processor already specified:***switch*

  This message is output if there is more than one **/T** switch in the command line of a **tf** command.

- **too many code fragments**

  The code fragment buffer is used by the code generator. This message is likely to happen only when the program contains extremely complex expressions. The cure is to expand the size of the code fragment buffer by using the **/B** switch (see section 17.2.9).

- † **Triad table overflow**

  The triad table is used by the syntactic analyser. If this error is reported, the cure is to expand the size of the triad table using the **/B** switch (see section 17.2.9).

- **/T4 or /T8 required**

  This message will be output if a **tf** command is given without a target processor specifying switch.

- **unable to access temporary file**

The compiler uses various temporary files during its execution. If the file system reports an error when it attempts to open one of these files, this message will be output.

- **`unable to create temporary file`**

  This message is output when the compiler is unable to create one of the temporary files it needs, because of a file system error. The most likely cause of this is a full disk.

- **`unable to open`** ***filename*** **`as listing file`**

  This message is output when the user requests a compilation listing, but the compiler is unable to open the listing file, either because of a file system error (e.g., a full disk) or because the filename was specified erroneously.

- **`unable to open`** ***filename*** **`as object file`**

  This message is output when the compiler is unable to open the object file (whether binary or hexadecimal), either because of a file system error (e.g., a full disk) or because the filename was specified erroneously.

- **`unable to open`** ***filename*** **`as source file`**

  The given *filename* has been specified in the command which invoked the compiler but such a file cannot be accessed. Check that the filename has been spelled correctly and that it exists in the relevant directory.

- **`unknown switch:`***switch*

  The *switch* included in the error message was present in the command line, but it is not a switch the compiler recognises.

- *value* `is outwith the range for` *parameter*

  This message indicates an error in a `/B` switch. The message will be accompanied by a table of the allowed values for the three parameters of the switch. Re-type the command with all the parameters in range.

- ... *reason* ... `; please submit a CSR`

  This message indicates a fault in the compiler itself. In some cases the *reason* may give a clue to a possible avoidance procedure but in all cases such messages should by reported to 3L by means of a *Customer Software Report* (CSR).

  If any other error messages have been generated before this fatal error message it is possible that a previous error has confused the compiler. Correcting the other errors may remove the cause of this message.

17.6.4 Run-Time Errors

17.6.4.1 General Input/Output Errors

Errors sometimes occur in the input/output routines of the run-time library. These result either from programming errors or from interactions with the operating system, and happen during the processing of one of the I/O statements (see chapter 16).

When such an error happens, the program will jump to the label given in the `ERR=` specifier of the I/O statement, if any (see section 16.3.1.1, for example). In addition (or alternatively), if the `IOSTAT=` specifier has been given, the error number will be placed in the specified variable. In neither of these cases will an error message be output. If neither of these two specifiers is given, however, a message will be output, all the currently open units will be closed, and the program will then be terminated.

Notice that if an I/O error happens in a subsidiary thread (that is, within a thread that has been invoked by `F77_THREAD_START` or `F77_THREAD_CREATE`) the program will not be terminated in the usual way. Instead, after the usual error message has been output, there will be a condition-handling error (see section 17.6.4.4).

The format of an I/O error report is as follows:

```
Input/Output Error nnn: message text

   In Procedure: procedure name
        At Line: source line number

      Statement: I/O statement type
           Unit: unit identifier
   Connected To: file name
           Form: Formatted or Unformatted
         Access: Sequential or Direct
        Nextrec: record number
Records Read    : number of records input
Records Written: number of records output
Current IO Buffer: snapshot
FORTRAN ERROR
```

nnn is the I/O error number and *message text* the message: a list of these can be found in appendix I. The *unit identifier* will be a number, or the words `Internal File`, in the case of an internal file (see section 16.7). The *snapshot* will show the contents of the current record with a pointer to the current position within it. Not all the above information will be supplied for every error.

I/O error report example

```
Input/Output Error 153: Input file ended

   In Procedure: MAIN PROGRAM
        At Line: 1

      Statement: Formatted READ
           Unit: 5
   Connected To: Standard Input Stream 0
           Form: Formatted
```

```
          Access: Sequential
Records Read   : 3
Records Written: 0
FORTRAN ERROR
```

17.6.4.2 Errors in Run-Time Formats

Normally, formats are specified in I/O statements by reference to the label of a **FORMAT** statement. Sometimes, however, the format may be held in a character variable, or an array, or may be specified as a character expression. In these cases, coding errors in the format specification will not be noticed until run time.

Errors of this kind are controlled by the **ERR=** and **IOSTAT=** specifiers of the I/O statements, in the same way as general I/O errors are. If this is not done, however, a message will be output, all the currently open units will be closed, and the program will then be terminated.

The format of this kind of error message is as follows:

```
Format Error nnn : text
Current Format:
format
  !
FORTRAN ERROR
```

nnn is an error number, and *text* is message describing the error. A list of error numbers and their corresponding messages may be found in appendix I. *format* is the text of the format specification, and the '**!**' points at the character which caused the error.

17.6.4.3 Errors Returned by `afserver`

If the `afserver` is unable to perform a function when the run-time library requests it, it reports an error condition. Normally, the run-time library anticipates such error conditions, and they are reported in the usual way, using the format described above in section 17.6.4.1.

In certain circumstances, however, the **afserver** may return an unexpected error code, and these are reported to the user in the same way as described in section 17.6.4.1, except for the first line, which has this format:

Alien Filer Error *nnn*: *message text*

where *nnn* is the **afserver**'s error code, and *message text* is an explanatory message. A list of the error codes reported and their corresponding messages may be found in appendix I. The 3L publication *Technical Note 3*[18] contains a complete list of **afserver** error codes.

17.6.4.4 Exceptional Errors

In certain exceptional circumstances, the run-time library may be unable to recover sensibly from an error condition. Three types of error message may result from this.

Tracebacks

The traceback message enables to user to discover the subprogram where the error has taken place, and the sequence of subroutine or function calls which has lead to it. It starts with a line of this format:

%event *p q r*

The three numbers *p*, *q* and *r* identify the kind of error which has occurred. They refer to events within the run-time library, and so in most cases will not be very helpful, though the following explanations of the meaning of *p* may be of some use:

0	Program is terminating
1	Overflow or truncation
2	Not enough of a resource: usually memory

3	Data transmission error
4	Invalid data, e.g., badly formatted number
5	Invalid argument to run-time library control structures, or corrupt internal data structures.
6	Value out of range
7	Error in run-time library's string manipulation primitives
8	Undefined value
9	Input/output error
10 - 15	Undefined

After this, the run-time library outputs a number of lines of the following format, each of which specifies a level of subprogram calling, the lowest being output first.

hexnum module subprogram

where:

module	is a module name. Normally this is the name of the source file containing the subprogram.
subprogram	is the name of the subprogram involved.
hexnum	is a location relative to the start of the code for that subprogram. In the case of the first line, it denotes a place near to where the error happened. In the other lines, it denotes the return address from the call to the subprogram in the line above.

The top few lines may contain module and subprogram names which are unfamiliar to you; these denote parts of the run-time library.

Condition-Handling Errors

This message is output when the condition-handling software itself (which is responsible for the traceback output described above) cannot continue. One important circumstance which can bring this about is if input-output errors happen as a result of input-output statements used in a subsidiary thread, that is, in a subroutine which has been invoked via `F77_THREAD_START` or `F77_THREAD_CREATE` (see section 18.2.3). It can also happen if a `STOP` statement is used within a subsidiary thread (where a call to `F77_THREAD_STOP` should be used instead).

The format of the message is:

```
signal: error n
```

Static Data Overflow

If the amount of static data required by a program exceeds the amount of memory available, the run-time library will output the following message:

```
static space too small
```

The uses of static storage are explained in section 3.5. Note that tasks in configured applications will not normally be able to present this message to the user, and will simply "seize up" if this condition arises.

Chapter 18

The Parallel Fortran Run-Time Library

18.1 Purpose of the Run-Time Library

The Parallel Fortran run-time library is a collection of compiled subprograms which perform functions needed by most Fortran programs. Those subprograms needed by any particular program are incorporated in that program by the linker (see section 3.3).

The subprograms included in the run-time library are of three kinds.

1. Those used to perform some of the commonly-used operations of the language itself, such as input and output, scanning formats, opening and closing files, and so on. These will not be discussed explicitly here, since they are not directly called by the programmer.

2. Intrinsic functions. These are discussed in section 14.1.2.1, and a complete list may be found in appendix E.

3. Non-intrinsic functions and subroutines. These are discussed in the following section.

18.2 Non-Intrinsic Subprograms

18.2.1 Conventions

18.2.1.1 Package Files

In order to make use of the various functions and subroutines described in the rest of this chapter, the programmer will have to write various declarative statements. These include type declarations for functions, and in certain cases **PARAMETER** statements to define constants. To make this easier, the available functions and subprograms have been divided into groups called *packages*, and for each package a file called a *package file* has been supplied. By using the **INCLUDE** statement (see section 14.7) to include the appropriate package file, the programmer is enabled to use the subprograms in a package without further red tape.

The package files should be installed in directory **\tf2v1** along with the rest of Parallel Fortran (see chaper 1). This means that in most cases a package file can be included by a statement of this format, which should be coded immediately after any **IMPLICIT** statements:

```
INCLUDE 'package-name.INC'
```

where *package-name* is the name of one of the packages described below. The appropriate **INCLUDE** statement is needed in every subprogram which uses the package. Full details of how the **INCLUDE** statement is evaluated may be found in section 17.3.

18.2.1.2 Format of Synopses

The sections below describe each of the subprogram packages in turn. The description of each subprogram starts with a synopsis, which takes the form of a sequence of Fortran statements.

1. In the case of functions, an explicit type statement is used to show what is the type of the function. For example:

```
INTEGER F77_ALLOC_HOST_MEM
```

   This statement need not be coded: it is in the package file.

2. Next, there follow explicit type statements for each of the parameters of the subprogram, and in the case of functions, for a variable to receive the function's value. For example:

```
INTEGER NBYTES
```

   These statements do not necessarily have to be coded. They are included in the synopsis for your information only.

3. Finally, an example of how the subprogram is to be called, either as a function or a subroutine. For example:

```
I = F77_ALLOC_HOST_MEM(NBYTES)
CALL F77_FREE_HOST_MEM(I)
```

18.2.2 The DOS Package

These functions and subroutines allow a program running on a transputer system which is hosted by an MS-DOS computer to access the software interrupts, DOS function calls and the memory of the host system. Further discussion and examples of the use of these subprograms may be found in section 3.6.

All subprograms using these routines should include the appropriate package file, thus:

```
INCLUDE 'DOS.INC'
```

Several of these subprograms make use of a parameter known as a *DOS block*. This takes the form of an integer array, the length of which is the value of the constant F77_DOS_BLOCK_SIZE, defined in DOS.INC. The elements of the DOS block correspond to 16-bit registers of the MS-DOS machine. The following constants are defined in

the package file to enable the programmer to access these elements using the familiar Intel names:

```
F77_DOS_AX  F77_DOS_BX  F77_DOS_CX  F77_DOS_DX
F77_DOS_SI  F77_DOS_DI
F77_DOS_CS  F77_DOS_DS  F77_DOS_SS  F77_DOS_ES
F77_DOS_CFLAG
```

Each of these elements is a 32-bit integer, the low-order 16 bits of which are used to hold the register contents. When a DOS block is sent to MS-DOS, the high-order 16 bits are ignored. In DOS blocks sent back from MS-DOS, the high-order 16 bits are always 0. The **F77_DOS_CFLAG** field returns the value of the C flag after an interrupt.

F77_ALLOC_HOST_MEM Allocate host memory

```
INTEGER F77_ALLOC_HOST_MEM
INTEGER NBYTES, IADDR
IADDR = F77_ALLOC_HOST_MEM (NBYTES)
```

This function allocates a block of at least **NBYTES** bytes in the base memory of the MS-DOS host computer and returns its 32-bit address. If the memory cannot be allocated, the value 0 is returned. The allocated memory cannot be accessed directly by the transputer program; rather, data can be moved between the transputer system and the host by means of the subroutines **F77_BLOCK_TO_HOST** and **F77_BLOCK_FROM_HOST**.

Note that the Intel 80*x*86 architecture limits the amount of memory which can be contained in a single segment to 65536 (10000_{16}) bytes; **F77_ALLOC_HOST_MEM** cannot allocate more than this architectural limit.

F77_FREE_HOST_MEM Free host memory

```
INTEGER IADDR
CALL F77_FREE_HOST_MEM (IADDR)
```

This subroutine frees the host memory allocated by a previous call on **F77_ALLOC_HOST_MEM**. The parameter **IADDR** should contain the 32-bit address of the memory block.

F77_BLOCK_TO_HOST Transfer memory block to host

```
INTEGER IADDR, NBYTES
CALL F77_BLOCK_TO_HOST (OURS, IADDR, NBYTES)
```

This subroutine transfers NBYTES of data from the transputer's memory, starting at OURS, to the host's memory, starting at the location whose 32-bit address is held in IADDR. The parameter OURS is the name of any Fortran variable, array or array element. The host memory block will normally have been allocated by F77_ALLOC_HOST_MEM.

F77_BLOCK_FROM_HOST Transfer memory block from host

```
INTEGER IADDR, NBYTES
CALL F77_BLOCK_FROM_HOST (IADDR, OURS, NBYTES)
```

This subroutine transfers NBYTES of data from the host's memory, starting at the location whose 32-bit address is held in IADDR, to the transputer's memory, starting at OURS. The parameter OURS is the name of any Fortran variable, array or array element. The host memory block will normally have been allocated by F77_ALLOC_HOST_MEM.

F77_READ_SEGMENTS Read host segment registers

```
INTEGER DOSBLOCK (F77_DOS_BLOCK_SIZE)
CALL F77_READ_SEGMENTS (DOSBLOCK)
```

This subroutine reads the current values of the host 80x86 processor's segment registers into DOSBLOCK, which is a DOS block as described above. Only the segment register elements of the DOS block array are changed.

F77_HOST_INTERRUPT Perform host interrupt

```
INTEGER INTNO, DOSBLOCK(F77_DOS_BLOCK_SIZE)
LOGICAL SEGS
CALL F77_HOST_INTERRUPT (INTNO, SEGS, DOSBLOCK)
```

This subroutine loads the contents of DOSBLOCK into the host's registers, and then calls host interrupt INTNO. DOSBLOCK should

be a DOS block, as defined above. If SEGS has the value .FALSE., the host's segment registers are not changed, that is, the segment register elements of DOSBLOCK are not used. If SEGS has the value .TRUE., however, the host's segment registers are loaded from DOSBLOCK as well.

The contents of the host's registers after the interrupt has completed are returned to DOSBLOCK. This includes the contents of the host's segment registers, whatever the value of SEGS. The original contents of DOSBLOCK are lost.

F77_INP Read host I/O port

```
INTEGER F77_INP
INTEGER IPORT, IVAL
IVAL = F77_INP (IPORT)
```

The F77_INP function reads a value from one of the host PC's input ports. The argument IPORT specifies the port which is to be read. The value read is returned as the value of the function.

F77_OUTP Write to host I/O port

```
INTEGER IPORT, IVAL
CALL F77_OUTP (IPORT, IVAL)
```

F77_OUTP writes the low-order byte of the integer value given as its second argument to one of the host PC's output ports. The first argument specifies the address of the output port.

18.2.3 The THREAD Package

The subprograms in this package allow a Parallel Fortran program to create new threads of execution within a single task. Every subprogram which uses the package should include the package file:

```
INCLUDE 'THREAD.INC'
```

Each thread executing on a transputer has a priority, which is either "urgent" or "not urgent". The package file defines the following constants to represent these:

- F77_THREAD_URGENT
- F77_THREAD_NOTURG

An important point about threads in Parallel Fortran arises from the fact that a Fortran subprogram is not reentrant. Just as subprograms cannot call themselves (see section 14.1.3), so a subprogram cannot be called by more than one thread at a time. This applies both to the subroutines called directly by the THREAD subroutines, and to all subprograms called indirectly through them. If more than one thread needs to make use of a function or subroutine, it should be protected by a semaphore (see section 18.2.4) to ensure that it is only being executed once at any one time.

STOP and PAUSE Statements and EXIT Subroutine

The STOP and PAUSE statements and the EXIT subroutine should only be used in a program's main thread. Using them in a subsidiary thread will cause a condition-handling error (see section 17.6.4.4). See sections 13.5 and 13.6.

F77_THREAD_START Start a general thread

```
      EXTERNAL SUB
      INTEGER WSSIZE, FLAGS, NARGS
      CALL F77_THREAD_START (SUB, WSARRAY, WSSIZE, FLAGS,
     1      NARGS, ARG1, ..., ARGN)
```

This subroutine starts a new thread based on the subroutine SUB. The new thread uses the area starting at WSARRAYto hold its workspace; normally this would be an integer array. WSSIZE is the size of WSARRAY, in bytes. The new thread will stop either when it executes the subroutine F77_THREAD_STOP, or when SUB returns.

The FLAGS argument is a set of attributes for the new thread. At present, the only attribute available is the thread's priority, which should be either F77_THREAD_URGENT or F77_THREAD_NOTURG. Normally, new threads should be started at the same priority as the current thread. This is achieved by passing the result of the function F77_THREAD_PRIORITY described below as the value of this argument. Other than the priority specification, all bits in the FLAGS argument are reserved, and should be 0.

The arguments ARG1,...,ARGN will be passed on to the thread's subroutine SUB as its arguments. The number of arguments must be supplied in NARGS, and, so long as this is correct, there is no limit to the number of arguments which may be passed. The arguments may be of any type, and it is the responsibility of SUB to handle them correctly. Variables of type CHARACTER are passed as two arguments, and NARGS must take account of this. For example:

```
CHARACTER*5 HELLO
INTEGER CATS
:
CALL F77_THREAD_START (DOIT, WS, 10000, IPRIO, 3, HELLO,
CATS)
```

In this case, the two arguments HELLO and CATS are passed to the subroutine DOIT. However, as HELLO is a character variable, the value of NARGS has to be given as 3, not 2.

See also the description of F77_THREAD_CREATE, which simplifies thread creation by starting a thread at the current priority and allocating the thread's workspace from the heap.

F77_THREAD_CREATE Create a simple thread

```
LOGICAL F77_THREAD_CREATE
EXTERNAL SUB
INTEGER WSSIZE, NARGS
LOGICAL STATUS
```

```
STATUS = F77_THREAD_CREATE (SUB, WSSIZE,
1             NARGS, ARG1, ..., ARGN)
```

The subroutine **SUB** is started as a new thread, running at the same priority as the current thread, with a workspace of **WSSIZE** bytes. An attempt is made to take this workspace from the heap. If there is not enough space available, the **F77_THREAD_CREATE** returns the value **.FALSE.**; otherwise, it starts the thread and returns **.TRUE.**. This workspace is never returned to the heap and remains unavailable for future re-use; if this is likely to be a problem, it would be better to use the **F77_THREAD_START** subroutine.

The **NARGS,ARG1,...,ARGN** sequence is used to pass arguments to **SUB**, in the same way as for **F77_THREAD_CREATE**.

This function is a shorthand way of calling the more general subroutine **F77_THREAD_START** in the most usual circumstances. Note that it can only be used be one thread at a time, so if necessary, **F77_THREAD_USE_RTL** should be called first (see below).

F77_THREAD_STOP Stop the current thread

```
CALL F77_THREAD_STOP
```

This subroutine stops the current thread. The current thread is also stopped if its main subroutine returns.

F77_THREAD_PRIORITY Return the current thread's priority

```
INTEGER F77_THREAD_PRIORITY
INTEGER I
I = F77_THREAD_PRIORITY()
```

This function returns the priority of the current thread, which will be either **F77_THREAD_URGENT** or **F77_THREAD_NOTURG**.

F77_THREAD_RESTART Restart a thread

```
INTEGER HANDLE
CALL F77_THREAD_RESTART (HANDLE)
```

HANDLE should be a pointer to the workspace of the thread it is desired to restart. Currently, the only value that should be passed to F77_THREAD_RESTART is one produced by F77_CHAN_RESET (see section 18.2.6 below).

This subroutine can be used to restart threads which have been stopped because the channel on which they were attempting to communicate has been reset using a call to F77_CHAN_RESET.

F77_THREAD_DESCHEDULE Make the current thread momentarily unable to execute

```
CALL F77_THREAD_DESCHEDULE
```

This subroutine causes a thread to become momentarily unable to execute (usually for one timer tick); this will cause it to be *descheduled* from the processor, thus allowing some other thread to resume execution in its place. Eventually, the thread which called F77_THREAD_DESCHEDULE will resume.

This subroutine can be used by a thread performing some background computation to prevent it from "hogging" the processor to the detriment of other threads executing at the same priority level. In effect, a priority level even less urgent than F77_THREAD_NOTURG can be achieved for use by threads performing long-term CPU-intensive tasks whose results are not expected to be immediately required.

F77_THREAD_USE_RTL Reserve the RTL to the current thread

```
CALL F77_THREAD_USE_RTL
```

In a program in which many execution threads are active, access to the Parallel Fortran run-time library must be synchronised, so that only one thread may be performing a library operation at one time. If this is not done, the internal data structures of the run-time library are likely to be corrupted, with unpredictable results. The required synchronisation is achieved by the subroutines F77_THREAD_USE_RTL and F77_THREAD_FREE_RTL. If there is more than one thread

running, any thread which wishes to use one of the following statements must call F77_THREAD_USE_RTL first:

```
READ        WRITE       PRINT
OPEN        CLOSE       INQUIRE
ENDFILE     REWIND      BACKSPACE
PAUSE       STOP
```

The following library subprograms, described in this chapter, must also be protected in the same way:

- Any member of the DOS package
- F77_THREAD_CREATE
- ICLOCK
- F77_DO_COMMAND
- EXIT

Apart from these, intrinsic functions and the non-intrinsic subprograms described in this chapter may, however, be used without this precaution.

F77_THREAD_USE_RTL either reserves the run-time library for the current thread, or if it is already reserved by another thread, suspends the current thread until the run-time library is free.

F77_THREAD_FREE_RTL Free the RTL

```
CALL F77_THREAD_FREE_RTL
```

F77_THREAD_FREE_RTL is used in conjunction with F77_THREAD_USE to synchronise the use of the Parallel Fortran run-time library. When a thread has finished using the run-time library, it should call this subroutine to permit other threads to use it.

18.2.4 The SEMA Package

This group of subprograms allows a Parallel Fortran program to create and manipulate semaphores, which can be used to synchronise the activity of several concurrently executing threads. All subprograms using these subroutines should include the appropriate package file:

```
INCLUDE 'SEMA.INC'
```

A semaphore in Parallel Fortran is represented by an integer array of length F77_SEMA_SIZE, this being a constant defined in the package file. Normally, a semaphore is in a COMMON block, so that more than one thread can access it.

Note that any particular semaphore must be accessed only by threads executing at one particular priority. For example, it would be acceptable for a set of "urgent" threads to synchronise through a semaphore, or for a set of "not urgent" threads to do this, but not for a mixture of threads executing at different priorities. Threads executing at different priorities can synchronise by passing messages along channels (see section 18.2.6).

F77_SEMA_INIT Initialise a semaphore

```
INTEGER MYSEMA(F77_SEMA_SIZE), VALUE
CALL F77_SEMA_INIT(MYSEMA, VALUE)
```

This subroutine initialises the semaphore variable MYSEMA to an initial state in which:

- the queue of threads waiting for the semaphore is empty
- the value of the semaphore is VALUE.

Semaphores should always be initialised using F77_SEMA_INIT, since if this is not done the first F77_SEMA_SIGNAL or F77_SEMA_WAIT operation on the semaphore may cause the transputer system to behave unpredictably.

F77_SEMA_SIGNAL Perform a *signal* operation on a semaphore

```
INTEGER MYSEMA(F77_SEMA_SIZE)
CALL F77_SEMA_SIGNAL(MYSEMA)
```

If there are threads waiting for the semaphore **MYSEMA**, one of them will be chosen and made able to execute again. The value of the semaphore under these conditions will always be 0, and will remain unchanged.

If there are no threads waiting for **MYSEMA**, its value will be increased by 1.

F77_SEMA_SIGNAL_N Perform *n signal* operations on a semaphore

```
INTEGER MYSEMA(F77_SEMA_SIZE), N
CALL F77_SEMA_SIGNAL_N(MYSEMA, N)
```

This subroutine calls the subroutine **F77_SEMA_SIGNAL** *n* times, in sequence.

F77_SEMA_TEST_WAIT Test a semaphore

```
LOGICAL F77_SEMA_TEST_WAIT
INTEGER MYSEMA(F77_SEMA_SIZE)
LOGICAL L
L = F77_SEMA_TEST_WAIT(MYSEMA)
```

If the value of the semaphore **MYSEMA** is not zero, its value is decreased by one, and the value **.TRUE.** is returned.

Otherwise, if the value of the semaphore is zero, **F77_SEMA_TEST_WAIT** immediately returns the value **.FALSE.** The value of the semaphore is unchanged. This is in contrast to the **F77_SEMA_WAIT** subroutine, which in this case does not return at once, but pauses the thread until the semaphore is non-zero.

F77_SEMA_WAIT Perform a *wait* operation on a semaphore

```
INTEGER MYSEMA(F77_SEMA_SIZE)
CALL F77_SEMA_WAIT(MYSEMA)
```

If the value of the semaphore MYSEMA is not zero, its value is decreased by 1.

Otherwise, if the value of the semaphore is zero, the value is left unchanged and the current thread is added to the list of threads waiting for the semaphore, and paused. It will be resumed by some future call on F77_SEMA_SIGNAL.

Programs should not rely on any relationship between the order in which threads start to wait on a semaphore and the order in which they will be resumed. At present, threads are simply "pushed down" onto the list of waiting processes, so that the last thread to start waiting on a semaphore will be the first to be resumed.

F77_SEMA_WAIT_N Perform *n wait* operations on a semaphore

```
INTEGER MYSEMA(F77_SEMA_SIZE), N
CALL F77_SEMA_WAIT_N(MYSEMA, N)
```

This subroutine calls the subroutine F77_SEMA_WAIT *n* times, in sequence. The calling thread may be forced to wait at any point in the sequence.

18.2.5 The TIMER Package

Each transputer associates a hardware timer with the group of threads executing at a particular priority. The following subprograms allow threads to manipulate the timer associated with the priority at which they are executing. Subprograms which use the TIMER package should include its package file:

```
INCLUDE 'TIMER.INC'
```

F77_TIMER_AFTER Compare two transputer timer values

```
LOGICAL F77_TIMER_AFTER
INTEGER T1, T2
```

```
LOGICAL L
L = F77_TIMER_AFTER(T1, T2)
```

This function returns .TRUE. if timer value T1 is *after* timer value T2, and .FALSE. otherwise.

F77_TIMER_DELAY Delay for some number of timer ticks

```
INTEGER TICKS
CALL F77_TIMER_DELAY(TICKS)
```

This subroutine causes the current thread to wait for at least TICKS ticks of the timer associated with the current thread's priority.

F77_TIMER_NOW Return the current timer value

```
INTEGER F77_TIMER_NOW
INTEGER I
I = F77_TIMER_NOW()
```

This function returns the value of the timer associated with the current thread's priority.

F77_TIMER_WAIT Wait until the current timer reaches some value

```
INTEGER ABSTIME
CALL F77_TIMER_WAIT (ABSTIME)
```

This subroutine causes the current thread to wait until the value of the timer associated with the the priority of the current thread is at least TIMER.

18.2.6 The CHAN Package

The subprograms described here allow programs to access the transputer's basic communication facility, which is to transfer a *message* across a *channel*. Every subprogram which makes use of the CHAN package should include its package file:

```
INCLUDE 'CHAN.INC'
```

Most of the following subprograms have a parameter named ICHANADDR in the synopses. This is an integer whose value is the address of a channel word. The programmer can find out the address of a channel word in one of the following ways:

- For channels which are bound to ports (see sections 5.1.2 and 26.2.9), the address of the channel word can be found by using the functions F77_CHAN_IN_PORT and F77_CHAN_OUT_PORT (see below). For example, the configuration file might include this statement:

```
connect ? cats[3] dogs[2]
```

  Task cats could send data to task dogs like this:

```
ICHANADDR = F77_CHAN_OUT_PORT (3)
CALL F77_CHAN_OUT_MESSAGE (length, buffer, ICHANADDR)
```

  Task dogs could receive data from cats like this:

```
ICHANADDR = F77_CHAN_IN_PORT (2)
CALL F77_CHAN_IN_MESSAGE (length, buffer, ICHANADDR)
```

- Any INTEGER variable may be used as an "internal" channel for communication between threads of the same task. The address of such a channel word may be found by using the F77_CHAN_ADDRESS function (see below). Note that an internal channel word must be initialised, using F77_CHAN_INIT, before it can be used.

- The following constants, defined in the package file, give the addresses of the channels associated with the Inmos links of the transputer on which the task is running.

F77_CHAN_LINK0INPUT	input channel associated with link 0
F77_CHAN_LINK0OUTPUT	output channel associated with link 0
F77_CHAN_LINK1INPUT	input channel associated with link 1
F77_CHAN_LINK1OUTPUT	output channel associated with link 1
F77_CHAN_LINK2INPUT	input channel associated with link 2
F77_CHAN_LINK2OUTPUT	output channel associated with link 2
F77_CHAN_LINK3INPUT	input channel associated with link 3
F77_CHAN_LINK3OUTPUT	output channel associated with link 3

- The package file supplies the constant **F77_CHAN_EVENTREQ**, which is the address of the channel associated with the transputer's external event mechanism.

The programmer should note that none of these routines are able to check the size of user-supplied buffers. The length of the data transfer depends on the length requested, either explicitly (as with **F77_CHAN_IN_MESSAGE**) or implicitly (in the sense that **F77_CHAN_IN_WORD** will transfer 4 bytes). In particular, if enough buffer space is not supplied to an input subroutine, adjacent memory will simply be overwritten, with unpredictable results.

F77_CHAN_ADDRESS Return the address of an internal channel word

```
INTEGER F77_CHAN_ADDRESS
INTEGER CHANWORD, ICHANADDR
ICHANADDR = F77_CHAN_ADDRESS (CHANWORD)
```

This function returns the address of its argument, for use when calling other subprograms in this package. **CHANWORD** should be an **INTEGER** variable or array element.

F77_CHAN_IN_BYTE Input a byte from a channel

```
INTEGER IBUFF, ICHANADDR
CALL F77_CHAN_IN_BYTE (IBUFF, ICHANADDR)
```

This subroutine reads a single-byte message from the channel whose address is **ICHANADDR** into the integer **IBUFF**. The byte of data is placed in the low-order byte of **IBUFF**; the other bytes are set to 0. (Compare this to **F77_CHAN_IN_MESSAGE** below).

F77_CHAN_IN_BYTE_T Input a byte from a channel, or timeout

```
LOGICAL F77_CHAN_IN_BYTE_T
INTEGER IBUFF, ICHANADDR, TIMEOUT
LOGICAL L
L = F77_CHAN_IN_BYTE_T (IBUFF, ICHANADDR, TIMEOUT)
```

This function attempts to read a single-byte message from the channel whose address is **ICHANADDR** into the integer **IBUFF**. If the communication does not take place within **TIMEOUT** ticks of the timer associated with the priority of the current thread, the function will terminate and return **.FALSE.**. If the communication succeeds within the timeout interval, the function will return **.TRUE.**

The byte of data is placed in the low-order byte of **IBUFF**; the other bytes are set to 0. (Compare this to **F77_CHAN_IN_MESSAGE_T** below).

F77_CHAN_INIT Initialise a channel word

```
INTEGER ICHANADDR
CALL F77_CHAN_INIT (ICHANADDR)
```

This subroutine initialises the channel word whose address is **ICHANADDR**. This operation consists of writing a special value into the channel word; the package file defines the constant **F77_CHAN_NOTPROCESS_P** with this value, which indicates that no threads are currently attempting to communicate through the channel.

All "internal" channel words must be initialised before the first attempt to communicate through them. If this is not done, the first attempt to communicate through the channel will cause the transputer processor to crash.

Note that the channel words bound to a program's input and output ports are already initialised by the calling environment, and should not be initialised again by the program.

F77_CHAN_IN_MESSAGE Input a message from a channel

```
INTEGER LENGTH, ICHANADDR
CALL F77_CHAN_IN_MESSAGE (LENGTH, BUFF, ICHANADDR)
```

This subroutine reads a message of length **LENGTH** bytes from the channel whose address is **ICHANADDR** into **BUFF**. **BUFF** may be any variable, array or array element. Notice that exactly the number of bytes specified in **LENGTH** will be changed. For example, if you specify a **LENGTH** of 1, and a **BUFF** of type **INTEGER**, the low-order byte only will be changed; the other bytes will remain unchanged.

F77_CHAN_IN_MESSAGE_T Input a message from a channel, or timeout

```
LOGICAL F77_CHAN_IN_MESSAGE
INTEGER LENGTH, ICHANADDR, TIMEOUT
LOGICAL L
L = F77_CHAN_IN_MESSAGE_T (LENGTH, BUFF, ICHANADDR,
TIMEOUT)
```

This function attempts to read a message of length **LENGTH** from the channel whose address is **ICHANADDR** into **BUFF**. **BUFF** may be any variable, array or array element. If the communication does not take place within **TIMEOUT** ticks of the timer associated with the priority of the current thread, the function will terminate and return **.FALSE.**. If the communication succeeds within the timeout interval, the function will return **.TRUE.**

F77_CHAN_IN_PORT Value of input port binding

```
INTEGER F77_CHAN_IN_PORT
INTEGER PORTNO, ICHANADDR
ICHANADDR = F77_CHAN_IN_PORT (PORTNO)
```

This function returns the *binding* of the specified input port. In most cases, this will be the address of the channel word to which the port is bound. Sometimes, however, a port is explicitly set to some literal value by means of the configurer

BIND statement (see section 26.2.11), in which case this value will be returned.

F77_CHAN_IN_PORTS Number of input ports

```
INTEGER F77_CHAN_IN_PORTS
INTEGER I
I = F77_CHAN_IN_PORTS()
```

This function returns the number of ports in the input port vector. This value is decided by the INS attribute of the configurer TASK statement; however, tasks which use the standard harness will always have 2 input ports.

F77_CHAN_IN_WORD Input a word from a channel

```
INTEGER ICHANADDR
CALL F77_CHAN_IN_WORD (WORD, ICHANADDR)
```

This subroutine reads a four-byte message from the channel whose address is ICHANADDR into the variable WORD. WORD may be any variable which is four bytes in length, typically an INTEGER, REAL or CHARACTER*4, or an element in an array of one of these types.

F77_CHAN_IN_WORD_T Input a word from a channel, or timeout

```
LOGICAL F77_CHAN_IN_WORD_T
INTEGER ICHANADDR, TIMEOUT
LOGICAL L
L = F77_CHAN_IN_WORD_T (WORD, ICHANADDR, TIMEOUT)
```

This function reads a four-byte message from the channel whose address is ICHANADDR into the variable WORD. WORD may be any variable which is four bytes in length, typically an INTEGER, REAL or CHARACTER*4, or an element in an array of one of these types.

If the communication does not take place within TIMEOUT ticks of the timer associated with the priority of the current thread, the function will terminate and return .FALSE.. If the com-

munication succeeds within the timeout interval, the function will return .TRUE.

F77_CHAN_OUT_BYTE Output a byte to a channel

```
INTEGER IVAL, ICHANADDR
CALL F77_CHAN_OUT_BYTE (IVAL, ICHANADDR)
```

This subroutine sends a single-byte message consisting of the value IVAL to the channel whose address is ICHANADDR. Note that even if the high-order 3 bytes of IVAL are non-zero, only the low-order byte is sent.

F77_CHAN_OUT_BYTE_T Output a byte to a channel, or timeout

```
LOGICAL F77_CHAN_OUT_BYTE_T
INTEGER IVAL, ICHANADDR, TIMEOUT
LOGICAL L
L = F77_CHAN_OUT_BYTE_T (IVAL, ICHANADDR, TIMEOUT)
```

This function attempts to write a single-byte message whose value is IVAL to the channel whose address is ICHANADDR. If the communication does not take place within TIMEOUT ticks of the timer associated with the priority of the current thread, the function will terminate and return .FALSE.. If the communication succeeds within the timeout interval, the function will return .TRUE.

F77_CHAN_OUT_MESSAGE Output a message to a channel

```
INTEGER LENGTH, ICHANADDR
CALL F77_CHAN_OUT_MESSAGE (LENGTH, BUFF, ICHANADDR)
```

This subroutine sends a message of length LENGTH bytes from BUFF to the channel whose address is ICHANADDR. BUFF may be any variable, array or array element.

F77_CHAN_OUT_MESSAGE_T Output a message from a channel, or timeout

```
LOGICAL F77_CHAN_OUT_MESSAGE
INTEGER LENGTH, ICHANADDR, TIMEOUT
```

```
LOGICAL L
L = F77_CHAN_OUT_MESSAGE_T (LENGTH, BUFF, ICHANADDR,
TIMEOUT)
```

This function attempts to send a message of length LENGTH from BUFF to the channel whose address is ICHANADDR. BUFF may be any variable, array or array element. If the communication does not take place within TIMEOUT ticks of the timer associated with the priority of the current thread, the function will terminate and return .FALSE.. If the communication succeeds within the timeout interval, the function will return .TRUE.

F77_CHAN_OUT_PORT Return the value of an output port binding

```
INTEGER F77_CHAN_OUT_PORT
INTEGER PORTNO, ICHANADDR
ICHANADDR = F77_CHAN_OUT_PORT (PORTNO)
```

This function returns the *binding* of the specified output port. In most cases, this will be the address of the channel word to which the port is bound. Sometimes, however, a port is explicitly set to some literal value by means of the configurer BIND statement (see section 26.2.11), in which case this value will be returned.

F77_CHAN_OUT_PORTS Number of output ports

```
INTEGER F77_CHAN_OUT_PORTS
INTEGER I
I = F77_CHAN_OUT_PORTS()
```

This function returns the number of ports in the output port vector. This value is decided by the OUTS attribute of the configurer TASK statement; however, tasks which use the standard harness will always have 2 output ports.

F77_CHAN_OUT_WORD Output a word to a channel

```
INTEGER ICHANADDR
CALL F77_CHAN_OUT_WORD (WORD, ICHANADDR)
```

This subroutine sends a four-byte message from the variable WORD to the channel whose address is ICHANADDR. WORD may be any variable which is four bytes in length, typically an INTEGER, REAL or CHARACTER*4, or an element in an array of one of these types.

F77_CHAN_OUT_WORD_T Output a word to a channel, or timeout

```
LOGICAL F77_CHAN_OUT_WORD_T
INTEGER ICHANADDR, TIMEOUT
LOGICAL L
L = F77_CHAN_OUT_WORD_T (WORD, ICHANADDR, TIMEOUT)
```

This function attempts to send a four-byte message from the variable WORD to the channel whose address is ICHANADDR. WORD may be any variable which is four bytes in length, typically an INTEGER, REAL or CHARACTER*4, or an element in an array of one of these types.

If the communication does not take place within TIMEOUT ticks of the timer associated with the priority of the current thread, the function will terminate and return .FALSE.. If the communication succeeds within the timeout interval, the function will return .TRUE.

F77_CHAN_RESET Reset a channel

```
INTEGER F77_CHAN_RESET
INTEGER ICHANADDR, HANDLE
HANDLE = F77_CHAN_RESET (ICHANADDR)
```

This function resets the channel whose address is ICHANADDR. If the channel is associated with an Inmos link, then the hardware of that link is reset as well.

If a thread was attempting to communicate on the channel at the time of the reset, then a handle to that thread (which is now suspended) will be returned as the result of F77_CHAN_RESET. This handle can be used to restart the suspended thread at a later date by passing it to the subroutine F77_THREAD_RESTART (see section 18.2.3).

If the channel was idle at the time of the reset (i.e., if no thread was attempting to communicate on it) then the value F77_CHAN_NOTPROCESS_P will be returned.

18.2.7 The NET Package

The NET package consists of two subroutines for communication between the tasks of a processor farm (see chapters 7 and 27). Subprograms using these subroutines should include the package file:

```
INCLUDE 'NET.INC'
```

The NET subroutines differ from the CHAN routines in several respects.

- They do not specify a channel to use; in fact, the channels bound to input port 0 and output port 0 are always used.
- The actual destination of data sent via the NET subroutines is decided by the routing software: data sent by the master task are routed to a currently-idle worker task, and data sent by the worker tasks are routed back to the master task.
- NET messages do not have to be of fixed, predetermined lengths. The receiving task is informed how long a message is by the F77_NET_RECEIVE subroutine.

A *message* sent via the NET package consists of one or more *packets*. Each call on one of the NET subroutines sends or receives one packet. The package file defines the value of the constant F77_NET_MAX_PACKET_LENGTH, which is the limit on the size of packets. Messages which are longer than this must be broken into more than one packet, as explained below. In this case, the routing software guarantees that the component packets of a message will be received in the right order.

F77_NET_RECEIVE Receive a processor-farm message

```
INTEGER LENGTH
```

```
LOGICAL COMPLETE
CALL F77_NET_RECEIVE (LENGTH, BUFF, COMPLETE)
```

This subroutine receives a data packet from the processor-farm routing software. The master task will receive messages sent by worker tasks, and worker tasks will receive only messages sent by the master task.

The received packet is placed in **BUFF**, which may be any variable, array or array element. It need never be longer than **F77_NET_MAX_PACKET_LENGTH**, and may be shorter than this, if it is certain that no packet which is too long to fit will be received. The actual length of the received packet is placed in **LENGTH**.

If the received packet is the final or only packet in the message, **COMPLETE** is set to **.TRUE.**. Otherwise, it is set to **.FALSE.**, and the receiving task must call **F77_NET_RECEIVE** again, to receive the next packet.

F77_NET_SEND Send a processor-farm message

```
INTEGER LENGTH
LOGICAL COMPLETE
CALL F77_NET_SEND (LENGTH, BUFF, COMPLETE)
```

This subroutine will send a data packet to the processor-farm routing software. Messages sent by worker tasks are routed to the master task, and messages sent by the master task are routed to any idle worker task.

Data are sent from **BUFF**, which may be any variable, array or array element. **LENGTH** specifies the number of bytes to send, and may not be less than 0 or greater than **F77_NET_MAX_PACKET_LENG**
Messages longer than that must be broken into smaller packets. If **F77_NET_SEND** is called with **COMPLETE** set to **.TRUE.**, the packet is assumed to be the last or only packet in the message. All other packets should be sent with **COMPLETE** set to **.FALSE.**

F77_NET_BROADCAST Send a processor-farm broadcast

```
INTEGER LENGTH
CALL F77_NET_BROADCAST(LENGTH, BUFF)
```

This subroutine can be used by the master task to send a message to every worker task in the processor farm. It should not be used by any worker task.

The message to be sent is found starting at **BUFF**, which may be any variable, array or array element. The **LENGTH** argument specifies the length in bytes of the message. This subroutine is unlike **F77_NET_RECEIVE** and **F77_NET_SEND** in that the **LENGTH** argument is *not* restricted to **F77_NET_MAX_PACKET_LENGTH**. This means that the programmer does not have to split the message up into packets; this is done by **F77_NET_BROADCAST**. The worker tasks receive the message by calling **F77_NET_RECEIVE** in the usual way, possibly several times; **F77_NET_BROADCAST** ensures that the when the last packet is read, the **COMPLETE** argument is set to **.TRUE.** as usual.

F77_NET_BROADCAST can only be used when all the worker tasks are known to be idle. Typically, this would be at the beginning of the program run, before any work packets have been sent out. Later, the master task can broadcast new data, provided a result packet has been received corresponding to every work packet sent out.

18.2.8 The ALT Package

The **ALT** package is provided to enable the programmer to perform "guarded input", that is, to receive input from whichever of a group of channels is ready to transmit, if any. Subprograms which use the **ALT** package should include the package file:

```
INCLUDE 'ALT.INC'
```

F77_ALT_WAIT Wait till input is ready

```
INTEGER F77_ALT_WAIT
INTEGER NCHAN, ICHANADDR1, ..., ICHANADDRN, I
I = F77_ALT_WAIT (NCHAN, ICHANADDR1, ..., ICHANADDRN)
```

This function's arguments are a count of the possible channels for input (NCHAN) and a list of channel word addresses. The function waits till one of these channels is ready for input, and then returns an index into this list, such that the leftmost (ICHANADDR1) has the index 1.

F77_ALT_NOWAIT Identify a ready input channel

```
INTEGER F77_ALT_NOWAIT
INTEGER NCHAN, ICHANADDR1, ..., ICHANADDRN, I
I = F77_ALT_NOWAIT (NCHAN, ICHANADDR1, ..., ICHANADDRN)
```

This function's arguments are similar to those of F77_ALT_WAIT. If any channel is ready for input, the function returns its index in the list. However, if no channel is ready, the function does not wait, but immediately returns the value 0.

F77_ALT_WAIT_VEC Wait till input is ready (vector)

```
INTEGER F77_ALT_WAIT_VEC
INTEGER NCHAN, ICHANADDRARRAY(NCHAN), I
I = F77_ALT_WAIT_VEC (NCHAN, ICHANADDRARRAY)
```

ICHANADDRARRAY is an array of channel word addresses, and NCHAN is its length. The function waits till one of the channels whose addresses appear in the array is ready to input, and then returns the subscript of the element in the array which contains that channel's address.

F77_ALT_NOWAIT_VEC Identify a ready input channel (vector)

```
INTEGER F77_ALT_NOWAIT_VEC
INTEGER NCHAN, ICHANADDRARRAY(NCHAN), I
I = F77_ALT_VEC (NCHAN, ICHANADDRARRAY)
```

This function is similar to F77_ALT_WAIT_VEC except that if when the function is called there are no channels ready for input, the function does not wait, but returns the value 0.

18.2.9 Compatibility Subroutines

The subroutines discussed in this section are supplied only for compatibility with previous versions of the run-time library. New programs should use the **CHAN** package (see section 18.2.6) to perform channel operations. The compatibility subroutines do not have a package file.

These subroutines can only be used to perform input/output on the channels which are bound to ports. Consequently, the channel to use is specified not by a channel-word address, as is the case for the **CHAN** package, but by a port number. This must be in the range defined in the configuration file by the INS and OUTS attributes of the task (see sections 5.1.2 and 26.2.8). For example:

```
task mangle ins=3 outs=1
```

This line in the configuration defines input ports 0 to 2, and output port 0 only.

CHANINMESSAGE Receive a message from a channel

```
INTEGER IPORTNO, NBYTES
CALL CHANINMESSAGE (IPORTNO, BUFF, NBYTES)
```

This subroutine will input a message from the channel bound to port IPORTNO, and store it in BUFF. BUFF may be any variable, array or array element. NBYTES specifies the number of bytes to receive.

CHANOUTBYTE Send a byte to a channel

```
INTEGER IPORTNO
CALL CHANOUTBYTE (BUFF, IPORTNO)
```

This subroutine sends a single byte of data from BUFF to the channel which is bound to port IPORTNO. BUFF may be any variable, array or array element. In the case of an object of type integer, the low-order byte of the integer will be sent, and in the case of an object of type character, the left-most character will be sent.

CHANOUTMESSAGE Send a message to a channel

```
INTEGER IPORTNO, NBYTES
CALL CHANOUTMESSAGE (IPORTNO, BUFF, NBYTES)
```

This subroutine will send a message from BUFF to the channel bound to port IPORTNO. BUFF may be any variable, array or array element. NBYTES specifies the number of bytes to send.

CHANOUTWORD Send a word to a channel

```
INTEGER IPORTNO
CALL CHANOUTWORD (BUFF, IPORTNO)
```

This subroutine sends a word of data (that is, four bytes) from BUFF to the channel bound to port IPORTNO. BUFF may be any variable, array or array element.

18.2.10 Miscellaneous

This section describes subprograms for performing a number of miscellaneous tasks. There is a package file, MISC.INC, but not all the subprograms need it; for the ones that do, the appropriate INCLUDE statement is included in the synopses below.

18.2.10.1 Time

ICLOCK Get host clock time

```
INTEGER SECS
CALL ICLOCK (SECS)
```

This subroutine will place in SECS the elapsed time in seconds since 00:00:00 GMT on January 1st, 1970. ICLOCK depends on the **afserver** for this information, which has three effects:

- A call to ICLOCK must be protected be protected by calling F77_THREAD_USE_RTL first (see section 18.2.3).
- ICLOCK is not available in tasks which ahve been linked with the stand-alone library.
- The PC software upon which ICLOCK depends attempts to give you the time *in GMT*. Unless you tell the system otherwise, it assumes you are in the Pacific Standard Time zone of the USA and adjusts the value it returns accordingly. To make it aware of what time zone you are in, you can define the MS-DOS environmental variable TZ. For example, if you live in Great Britain, you could define TZ like this:

```
C>set tz=GMT
C>set tz=GMT1BST          during Summer Time
```

18.2.10.2 Not-a-Number and Infinity

The following functions test whether a REAL or DOUBLE PRECISION value is a NaN, $+\infty$ or $-\infty$, that is, Not-a-Number or Positive or Negative Infinity. These special values, as defined by the IEEE standard, are discussed in more detail in section 11.1.10. The term "finite" is used to describe a value which is not a NaN, $+\infty$ or $-\infty$.

F77_R_IS_NAN Test real for NaN

```
INCLUDE 'MISC.INC'
LOGICAL F77_R_IS_NAN
REAL R
LOGICAL L
L = F77_R_IS_NAN (R)
```

The function returns the value .TRUE. if the value R is a Not-a-Number; otherwise, it returns .FALSE.

F77_R_IS_FINITE Test real for finite

```
INCLUDE 'MISC.INC'
LOGICAL F77_R_IS_FINITE
REAL R
LOGICAL L
L = F77_R_IS_FINITE (R)
```

The function returns the value .TRUE. if the value R is finite; otherwise, it returns .FALSE.

F77_D_IS_NAN Test double for NaN

```
INCLUDE 'MISC.INC'
LOGICAL F77_D_IS_NAN
DOUBLE PRECISION D
LOGICAL L
L = F77_D_IS_NAN (D)
```

The function returns the value .TRUE. if the value D is a Not-a-Number; otherwise, it returns .FALSE..

F77_D_IS_FINITE Test double for finite

```
INCLUDE 'MISC.INC'
LOGICAL F77_D_IS_FINITE
DOUBLE PRECISION D
LOGICAL L
L = F77_D_IS_FINITE (D)
```

The function returns the value .TRUE. if the value D is finite; otherwise, it returns .FALSE.

18.2.10.3 Memory Access

The following subprograms provide facilities for "low-level" access to memory locations. Users should be aware that if these are misused, the results are unpredictable, but could be serious.

F77_PEEK_BYTE Peek a byte

```
INCLUDE 'MISC.INC'
INTEGER F77_PEEK_BYTE
```

```
INTEGER IADDR, IVAL
IVAL = F77_PEEK_BYTE (IADDR)
```

This function returns the value held in the byte at the absolute memory address IADDR. The value is returned as an unsigned (zero-extended) value.

F77_PEEK_WORD Peek a word

```
INCLUDE 'MISC.INC'
INTEGER F77_PEEK_BYTE
INTEGER IADDR, IVAL
IVAL = F77_PEEK_WORD (IADDR)
```

This function returns the value held in the word at the absolute memory address IADDR.

F77_POKE_BYTE Poke a byte

```
INTEGER IADDR, IVAL
CALL F77_POKE_BYTE (IADDR, IVAL)
```

The value of the argument IVAL is placed in the byte at absolute memory address IADDR. The value is silently truncated to the least-significant eight bits.

F77_POKE_WORD Poke a word

```
INTEGER IADDR, IVAL
CALL F77_POKE_WORD (IADDR, IVAL)
```

The value of the argument IVAL is placed in the word at absolute memory address IADDR.

%LOC Address of Variable

```
INTEGER %LOC, IADDR
IADDR = %LOC (VAR)
```

VAR may be any variable, array or array element; the function returns its absolute memory address.

F77_MOVE Block move

```
INTEGER NBYTES
CALL F77_MOVE (NBYTES, SOURCE, DEST)
```

The SOURCE and DEST arguments may be any variables, arrays or array elements. The subroutine moves the number of bytes specified by NBYTES from SOURCE to DEST.

F77_MOVE_A Block move absolute

```
INTEGER NBYTES, SRCADR, DSTADR
CALL F77_MOVE (NBYTES, SRCADR, DSTADR)
```

In this case, the two arguments SRCADR and DSTADR specify absolute memory addresses. The subroutine moves the number of bytes specified by NBYTES from SRCADR to DSTADR.

18.2.10.4 System Operations

F77_GET_COMMAND Get command parameters

```
CHARACTER*n BUFF
CALL F77_GET_COMMAND (BUFF)
```

The command parameter string is placed in BUFF. If the string is shorter than the argument, it is padded with spaces; if it is longer, it is truncated.

F77_DO_COMMAND Execute a host command

```
CHARACTER*n BUFF
CALL F77_DO_COMMAND (BUFF)
```

The argument string is submitted to DOS to be executed as a command. For example:

```
CALL F77_DO_COMMAND ('dir *.f77')
```

You should not try to execute a command which involves running another program on the transputer board; this would

have unpredictable results. Only one thread should use this subroutine at a time; this means that if necessary, F77_THREAD_USE_RTL should be called first (see section 18.2.3 above).

EXIT Terminate the program

```
INTEGER STATUS
CALL EXIT (STATUS)
```

If the task is linked with the full run-time library, all its files are closed, and the task is terminated. The value of STATUS is returned to DOS as a result code. EXIT should never be called from a subroutine which has been invoked in a subsidiary thread; if this is done, there will be a condition handling error (see section 17.6.4.4).

EXIT may also be called from a task linked with the standalone run-time library. In this case, it just stops the current thread, as F77_THREAD_STOP does; STATUS is ignored.

Chapter 19

The Linker

The linker utility, `linkt`, is compatible with all versions of the 3L compilers for C, Fortran and Pascal. It can be used in place of the linker distributed with earlier versions of these compilers. The linker is also compatible with Tbug, 3L's interactive source-level debugger.

The linker's function is to create an executable file from a number of object files. It can also be used to create libraries of object modules, which may themselves be searched by the linker when it is creating executable files.

19.1 Command Line

The linker is invoked by the command `linkt`. This command is followed by an ordered list of items giving the names of the object files and libraries to be linked, the name to be used for the executable file, and switches to control the linking operation.

The name of the executable file must be separated from the object file names by a comma ','; each object file may be separated from the next by either a space or a plus sign, '+'. Switches all start with

a slash, '/', and so do not need to be separated one from another, but spaces may be inserted between them for clarity.

The following are all valid examples of link commands.

```
C>linkt prog.bin library.bin,prog.b4
C>linkt prog.bin+library.bin, prog.b4
C>linkt prog1.bin prog2.bin lib.bin, myprog.b4
C>linkt prog1.bin+prog2.bin+lib.bin, myprog.b4
C>linkt prog.bin lib.bin, myprog.b4 /Q/Smyprog.map/Okernel
C>linkt prog.bin lib.bin, myprog.b4 /Q /Smyprog.map /Okernel
```

The order of the object file names in the link command is used to order the placement of the information they contain in the resulting executable file. Often this ordering is of no interest but it can be used to improve the performance of programs. This subject is discussed further in section 19.6.

19.2 File Name Conventions

In order to simplify commands, the linker will insert file name extensions where none has been given. If an explicit extension has been given it will be used without change.

The actual extension that will be appended to a file name depends on the sort of file being identified. The following table gives each sort of file known to the linker along with the appropriate extension.

executable file	.b4	object file	.bin
indirect file	.dat	optimization file	.opt
input library file	.bin	output library file	.lib
map file	.map		

As a result the examples given previously would have the identical effect if written in the following ways:

```
C>linkt prog library,prog
C>linkt prog+library, prog
C>linkt prog1 prog2 lib, myprog
```

```
C>linkt prog1+prog2+lib, myprog
C>linkt prog lib, myprog /Q/Smyprog/Okernel
C>linkt prog lib, myprog /Q /Smyprog /Okernel
```

19.3 The Output File

The output from a linking operation is usually a file containing a complete program in a form ready for execution. This file is called an *executable file*. The output may also be a *library* suitable for input to a subsequent link operation. Section 19.5 describes libraries.

The name for the output file is either specified explicitly on the command line (as in all the examples so far) or is inferred by the linker from the name of the *first* object file (or library file) seen, by removing any extension and then appending the extension `.b4`.

For example, each of the following commands generates an executable file named `test.b4`:

```
C>linkt test.bin fns.bin lib.bin, test.b4
C>linkt test fns lib, test
C>linkt test fns lib
C>linkt fns test lib, test
```

19.4 Indirect Files

It is quite common for programs to be built from a large number of object files, perhaps more than can comfortably be fitted into a single `linkt` command line.

The linker addresses this problem with *indirect files*, each of which contains one or more file names on separate lines. Indirect files may be given wherever object files are expected and the file names they contain are interpreted as the names of object files to be included in the linking operation.

In linker command lines, indirect files are always marked with the symbol '**@**' to distinguish them from other sorts of file. It is also possible to mark names within indirect files in this way. Such names are then taken to be the file names of nested indirect files. Indirect files may only be nested to a depth of 5.

For example, assume the file **list.dat** contains the following:

```
file1.bin
file2
file3.xxx
```

In the following example, the first four commands will all have the same effect, while the fifth command will generate an identical executable file but will write it to a file named **prog.b4**:

```
C>linkt @list
C>linkt @list, file1.b4
C>linkt file1 file2 file3.xxx
C>linkt file1 file2 file3.xxx, file1.b4
C>linkt @list, prog
```

Note that in the examples above, the first object file name in the indirect file will be the first object file seen by the linker and so it will be that file name which will be used, if necessary, to deduce the name for the output file.

Indirect files are also used to supply a list of optimization symbols to the linker. This is described in section 19.6.

19.5 Libraries

It is often convenient to be able to treat a group of object files as a single unit known as a *library file*. Accordingly, the linker provides the option of combining object files (and library files) into a new library file rather than the more usual executable file.

Once a library file has been generated it may be used wherever an object file is expected; unlike indirect files there is no need to mark the library file name in any way.

Library files have two advantages over indirect files. Firstly, moving a single library file to another place is simpler than moving many component object files and making sure that the corresponding indirect file is kept up to date. Secondly, accessing a single library file is faster than accessing an indirect file and several object files.

During the development of components which will eventually make up a library, indirect files may be more convenient as there will be no need to re-link the library whenever a component object file is changed.

The linker command to create a library is similar to that used to create an executable file, but includes the switch **/L**. When this switch is used the output file will be a library file and not an executable file. The name of the library file will be deduced, if necessary, in the same way as for executable files; that is, from the name of the first object file or library file found. The default extension **.lib** will be added if no extension is given.

The example below shows a graphics library being built from a core graphics module and two device driver modules. The library is then linked in the ordinary way with a user program. Indirect files are used to simplify the required linker commands.

```
C>type graflib.dat
core.bin
tek.bin
hp.bin

C>linkt @graflib,graflib/L

C>type myprog.dat
myprog.bin
graflib.lib
library.bin
harn.bin
```

```
C>linkt @myprog,myprog.b4
```

The switch /P can be used in place of /L and has exactly the same meaning.

The following switches are ignored when the /L or /P switches are used: /B, /C, /O, /S and /X. Section 19.9 contains a full description of the switches.

If the /G option is used when creating a library, any debug information present in the object files is passed through into the library. Otherwise this information is left out of the library.

19.6 The Executable Image

Unless otherwise instructed, the linker will place object files it has selected into the executable file in the order in which those object files were specified on the command line. This order is important if a program wishes to make use of the on-chip RAM.

When the on-chip RAM is used to hold programs, the code which has been placed at the beginning of an executable image is more likely to reside in RAM than code towards the end. Hence, in order to improve the performance of a program, the object file containing the code which is executed most frequently should be specified as the first object file in the link command.

In many cases, it may not be easy or possible to know which order to place the object files in. For example, the user may know which functions are executed most frequently, but not know which object files contain them, because they are part of a library. In this case, the user can specify a symbol to search for, and the linker will look for an object file which contains a definition of that symbol. Symbols used liked this are known as *optimization symbols*, and are specified by using the /O command line switch. Note that the switch uses the *letter* 'O' and not the *digit* '0'.

As an example, the following will place the object files which contain definitions of `fread` and `malloc` at the beginning of the resulting executable file `t.b4`:

```
C>linkt t library harn/Ofread/Omalloc
```

In this case, the object file containing the external symbol `fread` is placed at the start of the executable image. The object file containing the external symbol `malloc` is placed second in the executable image.

If an optimization symbol does not exist then the linker issues a warning. Sometimes the object file containing the symbol is not needed in the executable image; in other words, there are no references to it. In this case, if the object file is part of a library, the module is excluded from the executable image, and no warning is issued. If, on the other hand, the symbol is found in an object file named in the command line or in an indirect file, the object file is included in the executable image regardless.

Two or more optimization symbols may refer to the same object file, in which case the position of the object file will be determined by the position of the first symbol to refer to it.

After all optimization symbols have been processed and the object files which define optimized symbols have been placed at the start of the executable image, the linker will add the remaining object files to the executable image in the order they were found on the command line. In the previous example this would mean that object file `t` would be the third object file in the executable image and the object file `harn` would be the last.

It is often easier to place the list of optimization symbols in a file rather than keeping them on the command line. This may be done using indirect files in the same way as for object files except that the default extension is now `.opt`.

An example optimization file `optsyms.opt` might contain the following text:

```
fread
```

```
malloc
```

This file could then be used to optimize the position of the object files defining `fread` and `malloc` as in the following command:

```
C>linkt t library harn/O@optsyms
```

A warning is issued if the symbol is not defined in any of the object files.

19.7 Map Files

The linker can be requested to produce a *map file* which will contain a list of all the symbols (both code and data) that have been defined in the executable image. The map file will also contain information about the sizes of the code and static areas for each object file.

Map files are requested with the `/S` switch. By default, the name of the map file is derived automatically from the first object file name. In the following example a map file called `test.map` would be generated.

```
C>linkt test library harn/S
```

Alternatively, the map file name can be specified explicitly on the command line by placing a file name immediately after the `/S` as in the following example:

```
C>linkt test library harn/Smfile
```

The default extension `.map` will be added if no extension is given. The above example would create a map file called `mfile.map`.

19.8 Debug Tables

Object files created using the 3L compilers may contain information intended for use by Tbug, the 3L debugger[19]. By default, the linker

will discard this information in order to produce small executable files.

The switch /G will make the linker incorporate any debugging information present in the object files into the output file, which may be either an executable file or a library file.

19.9 Summary of Switches

The operation of the linker can be controlled by means of *switches*. Each switch starts with a slash character '/' and an identifying letter; it does not matter if this letter is given in upper case or lower case. The switches can be placed anywhere in the command line but they may not occur in indirect files. No spaces are allowed between a switch's identifying letter and the rest of the switch.

/B*file-name* This switch specifies that the file *file-name* is to be used in preference to the default bootstrap file. There is no default extension for *file-name*.

/C This switch stops the linker adding the bootstrap file to the executable file.

/G This switch results in the linker creating a debugger information area in the executable or library file.

/I This switch causes the linker to display its identity and along with various statistics about the executable file such as the code and static sizes and the maximum patch size used.

/L This switch makes the linker generate a library file rather than an executable file.

/O*optimization-symbol*
This switch gives priority to the position in the

executable image of the object file which defines *optimization-symbol*.

/O@*optimization-file*
This switch gives priority to the position in the executable image of the object files which define the symbols whose names are contained in the file *optimization-file*. The default extension for *optimization-file* is **.opt**.

/P This switch has the same effect has the **/L** switch.

/Q This switch suppresses all warning messages (see section 19.12).

/Q*n* This switch suppresses the output of message *n*. The number is one of those listed in appendix II. The purpose of this switch is explained in section 19.11 below.

/S This switch generates a map file taking its name from the first name in the list of object files.

/S*map-file* This switch generates a map file called *map-file*. The default extension for *map-file* is **.map**.

/X*entry-point*
This switch causes the linker to use the symbol *entry-point* in preference to **INMOS.ENTRY.POINT**, which is the default.

19.10 Using Batch Files

A batch file is a convenient way of calling the linker with the appropriate run-time library and harness. The linker accepts spaces between object file names, so the batch file can pass more than one parameter to the linker; unused parameters will be ignored. Switches can appear in any position on the command line, so they can be

passed as parameters to the batch file. For example, the batch file `tlink.bat` might look like this:

```
C>linkt %1 %2 %3 %4 %5 %6 %7 %8 %9 library.bin harn.bin
```

The following example shows how the batch file could then be used to link two files `file1.bin` and `file2.bin` into a library `file1.lib`:

```
C>tlink file1 file2/L
```

The batch file will then invoke the linker with the following command:

```
linkt file1 file2/L library harn
```

It is not possible include a comma in a batch file parameter. For this reason, you cannot explicitly pass an output file name to a batch file in its command line.

19.11 Duplicate Definitions

A duplicate definition occurs if two or more object files define the same symbol. The linker will issue a warning message about each occurrence of a duplicate definition and will use the first definition encountered. Object files are processed in the order in which they appear on the command line.

This facility can be useful when it is necessary to rewrite or alter an object file contained in a library. It can also be used to substitute one object file for another when creating a new library.

Occasionally, for example when several libraries are being used, it may be desirable to suppress a very large number of duplicate definition warning messages. This can be done by using the switch `/Q1`. This facility may be useful for OEM users of the linker.

19.12 Messages

The linker may issue one or more messages during a linking operation. These messages are used to draw the user's attention to unusual or incorrect situations.

There are two types of message: *warnings*, which indicate acceptable but possibly erroneous conditions, and *fatal errors* which result from conditions which are serious enough to terminate the linking operation.

A complete list of the messages output by the linker may be found in appendix H.

Chapter 20

The mempatch Utility

The linker program, `linkt`, normally produces an executable image file prefixed by a short bootstrap program which allows the `afserver` to load the image into an empty transputer: the bootstrap initialises the transputer and reads in the rest of the image file.

The bootstrap produced by the linker is designed to work with the Inmos B004 transputer board, or with an exact copy. These boards have either 1 or 2MB of RAM: the bootstrap may not work properly with partially B004-compatible boards which have different amounts of memory.

This problem does not affect task image files produced by the linker for use with the 3L configurers, since the configurers ignore any bootstrap code prefixed to the input task images and add their own bootstrap to the output application image file. The configurer-generated bootstrap can handle any amount of memory which is a multiple of 64KB.

The linker-generated bootstrap is only used if a single image file is run on its own on one transputer as described in chapter 3. In that case, the following problems may occur on a transputer board with other than 1 or 2MB of RAM:

- On systems with more than 2MB of memory, .b4 files produced by the linker will assume that only 2MB of memory is available; the program will not be able to take advantage of the rest of the physical memory in the configuration.

- On systems with less than 1MB of memory, .b4 files produced by the linker will assume too much memory is available, and are likely to fail when memory above the amount actually available is used.

- On systems with more than 1MB but less than 2MB of memory, one or other of the above effects will be observed, depending on the details of the board's address decoding hardware.

The **mempatch** utility allows you to modify .b4 files so that they will execute correctly with a particular memory configuration other than 1 or 2MB.

The compiler, linker and other utilities provided in this release all use the standard bootstrap, and may therefore require to be modified using **mempatch** if they are to be run on a transputer board with other than 1 or 2MB of RAM. Note that 3L does not guarantee that the compiler, linker and other programs will necessarily operate correctly if insufficient memory is available.

20.1 Identifying mempatch

If the **mempatch** utility is invoked without arguments, it will print identifying information similar to the following:

```
C>mempatch
usage: mempatch filename.b4 kilobytes
e.g.   mempatch myprog.b4 128
mempatch V1.2, Copyright (C) 1988, 3L Ltd.
```

A given version of **mempatch** can only be guaranteed to operate correctly with particular versions of the 3L high-level languages.

You should only use the version of **mempatch** supplied with this release in conjunction with the corresponding compiler and linker. **mempatch** will detect and reject any program image with which it is not compatible.

20.2 Invoking mempatch

The **mempatch** utility is invoked with a command line of the following form:

mempatch *image-file-name number-of-kilobytes*

For example, to patch the file **myprog.b4** for a system with only 64 kilobytes of memory, the following command line would be used:

```
C>mempatch myprog.b4 64
standard secondary bootstrap recognised
image now patched to 64 kilobytes
```

Note that the full filename of the program image file—including any **.b4** extension—must be supplied.

20.3 Re-invoking mempatch

A program image file may be patched more than once if, for example, available memory in the target system changes. The program file **myprog.b4** modified in the previous example might be modified again for a 128 kilobyte system as follows:

```
C>mempatch myprog.b4 128
previous patch value was 64 kilobytes
image now patched to 128 kilobytes
```

Chapter 21

The decode Utility

A separate decoder utility is supplied with Parallel Fortran which takes as its input the binary output file of the compiler, and produces a listing including both the source code and the disassembled machine code for each source line.

An example of **decode**'s output may be found in figure 21.1.

21.1 Usage

The decoder is started by a command of this format:

decode *filename*

Here, *filename* is the name of a binary output file from the compiler. If no extension is typed, **.bin** is assumed.

The decoder then attempts to find the source file, using the source file name given at compilation time, which is stored in the binary file. It applies this name in the context of the current directory when the decoder is run. Thus, if at compilation time the source file was specified as **down\cats**, and the current directory when the decoder is run

is **\mine**, the decoder will attempt to open **\mine\down\cats.f77** as the source file. The decoder should therefore be invoked with the current directory set to be the directory which was current when the file being decoded was compiled.

If **decode** cannot find the source file, it outputs a warning message and produces a disassembly listing without source lines.

The decoder's output is normally sent to the display. It may, however, be redirected or piped in the usual way, for example:

```
C>decode cats > cats.lis

C>decode cats | more
```

21.2 Features of the decode Program

The line **TOTALCODE 176 0** in the example reports the size of the program code for the module: in this case, 176 (decimal) bytes. The second number can be disregarded.

The line **STATIC 11** in the example reports the size of the static space required by the module, expressed this time in words (decimal).

Machine-code instructions are decoded into mnemonics. The decoder automatically merges **pfix**'s and **nfix**'s with the following opcode. There is full support for all T4 and T8 instructions, including the T8's '**fpu**' operations. Unrecognised indirect instructions are decoded as '**opr** *n*', and unrecognised **fpentry** instructions as '**ldc** *n*; **fpentry**'.

The destinations of **j** and **cj** instructions are shown as addresses in hexadecimal, rather than relative displacements. Calls to external symbols are shown symbolically if possible. The operand fields of all other direct instructions are shown in decimal.

The initialisation values of static data are shown in hexadecimal and ASCII.

The source code contents of files added to the program by means of `INCLUDE` statement files are not shown, but binary code generated from them is decoded and appears at the right point in the main source file.

21.3 Other Languages

The decoder can handle binary object files which are of the format described in the Inmos *Standalone Compiler Implementation Manual*[13]. As well as Parallel Fortran, the 3L Parallel C and Pascal compilers generate binary files of this kind, and these can therefore be decoded. If source files are available, the C or Pascal source program will be correctly included in the listing.

The Inmos stand-alone occam 2 compiler also generates binary files in this format, and should therefore be decoded correctly, although this cannot be guaranteed. The source programs are not shown, as the occam compiler does not generate the necessary line-number information.

The decoder cannot handle executable (`.b4`) files.

```
Transputer DECODE (V1.2) of decodex.bin
ID T4 "occam 2 V2.1" "F77_transputer V2.0"
SC 0
TOTALCODE 176 0
STATIC 11
REF #0,  "f_stop"
                          20  000AF                  | |
                    00000000  0004C                  |____|
  5845444F 4345440B 00000001  00098                  |.___.DECODEX|
                    3737462E  000A4                  |.F77|
    1        DATA I/3/
                    00000003  00070                  |.___|
              6E6961 6D5F6606  000A8                  |.f_main|
CODESYMB "f_main" 00000030
                          71  00030  ldl     1
                          30  00031  ldnl    0
                       20 20  00032  ldnl    MODNUM
                       BE 60  00034  ajw     -2
                          D0  00036  stl     0
    2        J = I + 5
                          70  00037  ldl     0
                          39  00038  ldnl    9
                          85  00039  adc     5
                          D1  0003A  stl     1
    3        END
                          40  0003B  ldc     0
                       4F 60  0003C  ldc     -1
                          70  0003E  ldl     0
           20 20 20 20 20 20  0003F  call    f_stop
                          B2  00045  ajw     2
                       F0 22  00046  ret
                       0D 61  00048  j       00037
                    FFFFFFFF  00078                  |....|
  00000000 00000000 01CB01CB  00000                  |....________|
  0000000B 00000000 00000000  0000C                  |________.___|
  0000005C 00000064 00000034  00018                  |4___d___\___|
  FFFFFFD4 0000001A 00000074  00024                  |t___.___....|
PATCH LONG 00000004 MODNUM
PATCH LONG 00000008 STATICFIX
PATCH LONG 0000000C INIT
PATCH LONG 00000010 LIMIT
                    FFFFFFFF  0007C                  |....|
  0000004A 00000008 00000030  00080                  |0___.___J___|
                    FFFFFFFF  0008C                  |....|
```

Figure 21.1: Example of output from decode

Chapter 22

The worm Utility

The **worm** utility is for exploring transputer networks. In its simplest form, it just counts the number of nodes in the network.

```
C>worm
one processor found
```

The **/F** option switch provides fuller information about each node, including:

- processor type (T414 or T800);
- processor clock speed;
- amount of external memory, in kilobytes (K);
- the number of *extra* processor cycles (penalties) required to access external memory as opposed to on-chip RAM (a minimum of two for a T414 or T800);
- the number of nodes through which work packets in a flood-configured application will be routed to get from the root transputer to this node. This number of "hops" may be greater

than the theoretical minimum imposed by the network configuration; it reflects the network spanning tree constructed by the flood-loading software.

On a single-processor system the output might look like this:

```
C>worm/f
one processor found
processor ROOT type=T414 20.0MHz, 3.0 penalties, 0 hops 2048K
   links to HOST[0],-------,-------,-------
```

The link connections from each node are listed from left to right in the order link 0, link 1, link 2, link 3. Here link 0 of the root transputer is connected to the host computer's link adapter and the other three links are unconnected. A dashed line, "-------" indicates an unconnected link.

The **/C** option makes the **worm** generate the node interconnection information in the form of a configuration file suitable for use with the static configurer.

```
! one processor found
processor HOST
processor ROOT
wire ? ROOT[0] HOST[0]
```

22.1 Notes

The **worm** will not discover "bare" nodes with little or no external memory. This is because the network loader on which it relies requires about 5–10KB of external RAM to function properly.

There may be a short delay before network information is displayed. This is because the **worm** waits for a certain amount of time before deciding that a link over which nothing is being received is unconnected and not just connected to a "slow" processor.

The **worm** writes its output on the standard output stream, normally the screen. Its output may be redirected to a file, or to a device

like a printer, using the DOS '>' facility. For example, to put a full description of a network into a file called **net.lis**:

```
C>worm/f >net.lis
```

Chapter 23

The tnm Utility

tnm shows the external symbols defined or referenced by an object file or library. For libraries, the names of the constituent object modules are also shown.

tnm is invoked like this:

tnm *filename*

The *filename* must be the name of an object (`.bin`) file produced by the compiler, or a library file produced by the linker. No default extension is supplied by **tnm**.

Object files and libraries are made up of sequences of object file records of various types. **tnm** scans the input file and writes (to standard output) the following types of record in a printable format. Other record types are skipped.

`COMPILER ID` records show the target processor (`T4` or `T8`) for which a module was compiled, and the version of the compiler used to compile it.

`LIBRARY` records delimit object modules within a library. They also contain the name of the following object module, except for occam

modules which do not have names and are therefore given numbers instead.

REF records name external symbols referred to by the current module. Note that simply referring to a symbol does not cause the module which defines it to be loaded. Only symbols which are actually used in "patch" records cause modules to be loaded. Patch records are not shown by **tnm**, because each symbol may be used in many places in an object file, requiring many patch records which would obscure the output produced.

CODE SYMBOL records define the locations of external symbols in the code area of the current module.

DATA SYMBOL records define the locations of external symbols in the static data area of the current module.

Figure 23.1 shows the start of the output produced by running **tnm** on the standard Fortran T4 run-time library.

The output from **tnm** normally appears on the screen, but it may be redirected to a file or device using the DOS '>' facility, like this:

```
C>tnm \tf2v1\frtlt4.bin >rtl.lis
```

```
LIBRARY MODULE 1: t4\pfaux.t4
   COMPILER ID occam 2 V2.1 IMP_transputer V1.3
   REF IMP_EVENT
   CODE SYMBOL f_cpystr
   CODE SYMBOL f_cpstr
   CODE SYMBOL f_concat
   CODE SYMBOL f_ibits
   CODE SYMBOL f_ibset
   CODE SYMBOL f_btest
   CODE SYMBOL f_ibclr
   CODE SYMBOL f_ishft
   CODE SYMBOL f_ishftc3
   CODE SYMBOL f_lle
   CODE SYMBOL f_lge
   CODE SYMBOL f_lgt
   CODE SYMBOL f_llt
   CODE SYMBOL f_len
   CODE SYMBOL f_index
   CODE SYMBOL f_ichar
   CODE SYMBOL f_char

LIBRARY MODULE 2: t4\pfcrts.t4
   COMPILER ID occam 2 V2.1 IMP_transputer V1.3
   REF REAL32TOREAL64
   REF REAL64TOREAL32
   REF REAL32EQ
   REF REAL32GT
   REF REAL64OP
   REF REAL32OP
   REF CSQRTP
   REF CEXPP
   REF CLOGP
   REF CSINP
   REF CCOSP
   REF SQRTP
   REF IMP_EVENT
   CODE SYMBOL f_csqrt
   CODE SYMBOL f_cexp
   CODE SYMBOL f_clog
   CODE SYMBOL f_csin
   CODE SYMBOL f_ccos
   CODE SYMBOL f_cabs
```

Figure 23.1: tnm Output

Chapter 24

The `tunlib` Utility

Individual object files can be extracted from a library using the `tunlib` command.

`tunlib` ***input-library output-library output-objfile symbol***

All four command line arguments are required. No default extensions are supplied by `tunlib`.

`tunlib` extracts an object module from the *input-library* and writes it to the *output-objfile*. The *input-library*, minus the extracted module, is copied to the *output-library*.

The module to be extracted is specified by giving the name of any external *symbol* it defines. Symbol matching is case sensitive. Note that Fortran symbols are converted to lower case before being output by the compiler, so the names for Fortran subprograms should be supplied to `tunlib` in lower-case form.

Do not use the same file name as both an input file and an output file. The effects of doing so are undefined.

In the example below, the module which defines the Fortran subroutine `PLOTPOINT` is extracted from a library called `graphlib.bin` and

written to an object file of its own called `point.bin`. The remainder of the library is written to a new file, `rest.bin`.

```
C>tunlib graphlib.bin rest.bin point.bin plotpoint
```

If we had wanted simply to delete the module containing `PLOTPOINT` from the library, we could have discarded the extracted object file by writing it to the null file, like this:

```
C>tunlib graphlib.bin newlib.bin nul plotpoint
```

`newlib.bin` is just `graphlib.bin` with the module which defined `PLOTPOINT` removed.

Chapter 25

The fpr Utility

The fpr program converts files formatted according to Fortran's carriage control conventions into files which may be printed under MS-DOS.

The program reads from standard input and writes to standard output, replacing the carriage control characters with characters that will produce the intended effect when the file is printed. According to the ANSI Fortran 77 standard, vertical spacing is determined by the first character of each line, and the standard uses the following table to define the effect of this character:

Character	Vertical Spacing Before Printing
␣	One Line
0	Two Lines
1	To First Line of Next Page
+	No Advance

If there are no characters in the record it is treated as a record consisting of a single space character. If the first character is not one of those mentioned above, it is treated as if it were a space character.

The following switches are recognised:

/B Output a form-feed at the beginning of the file; suppress the form-feed at the end.

/E Output a form-feed at the end, and not at the beginning. Of course, if the first character of the file is a '1' then a form-feed will be output in any case.

/N Do not output a form-feed either at the beginning or the end.

/I Print the program's identification.

The default behaviour is to output a form-feed at the end of the file. Note that if the output from **fpr** is sent to the screen of an IBM PC (or compatible) a form-feed will be displayed as a '♀' symbol.

fpr may be used to convert a file, like this:

```
C>fpr <output.dat >prntfile.dat
```

Alternatively, if a Fortran program writes to unit 6 (as preconnected), the output can be piped through **fpr**:

```
C>afserver -:b fortprog.b4 | fpr >prntfile.dat
```

Not For Sale

Chapter 26

Configuration Language Reference

The 3L configuration language is the language accepted by the various 3L configuration utilities. It is designed to allow easy description both of physical processor networks and of user applications built up out of tasks, without the user being concerned with the details of how the tasks are actually loaded into the processor network.

Each of the configuration utilities will, in general, accept a subset of the language described here, according to its needs. For example, the flood-fill configurer accepts the barest descriptions of the user tasks; it needs no description of the physical network because that information will be discovered at load time.

26.1 Standard Syntactic Metalanguage

In a formal description of a computer language, it is often convenient to use a more precise language than English. This language-description language is referred to as a *metalanguage*. The metalanguage which will be used to describe the configuration language is

that specified by British Standard 6154:1[7]. A tutorial introduction to the standard syntactic metalanguage is available from the National Physical Laboratory[8].

The BS6154 standard syntactic metalanguage is similar in concept to many other metalanguages, particularly those of the well-known Backus-Naur family. It therefore suffices to give a very brief informal description here of the main points of BS6154; for more detail, the standard itself should be consulted.

1. Terminal strings of the language—those not built up by rules of the language—are enclosed in quotation marks.

2. Non-terminal phrases are identified by names, which may consist of several words.

3. A sequence of items may be built up by connecting the components with commas.

4. Alternatives are separated by vertical bars ('|').

5. Optional sequences are enclosed in square brackets ('[' and ']').

6. Sequences which may be repeated zero or more times are enclosed in braces ('{' and '}').

7. Each phrase definition is built up using an equals sign to separate the two sides, and a semi-colon to terminate the right hand side.

26.2 Configuration Language Syntax

To simplify the explanation of the configuration language, the formal definition which follows in subsections 26.2.2 onwards deals only with the higher level syntax of the language. At this level, we can deal with how the significant characters of the language are built up into tokens and statements. The lower level syntax deals with the way in

which multiple input files are handled, with comments and with line continuation. This topic is treated informally in subsection 26.2.1.

The high level syntax given here has an additional simplification intended to make it more readable. To show this, consider the following syntax rule written in the BS6154 metalanguage:

example rule =
    "`first`", "`second`";

Interpreted strictly, this rule would be satisfied only by an input text which read "`firstsecond`". In the syntax presented here, it should be taken to match "`first`" *followed by* "`second`", but in such a way that the two items are distinguishable. For example, the two words here might be separated by a space character in the input file. When the two items are distinguishable in the input file without a space between them, then they may be abutted. This would be the case for the two items in the following example:

second example rule =
    "`first`", "`=`";

Valid input text for this rule could be, for example, "`first=`" or "`first =`".

26.2.1 Low Level Syntax

The general form of a configuration language "program" is designed to be as simple as possible to use.

The following example show the ways in which the formatting, commenting and continuation facilities available in the configuration language can be used:

```
! this is an example of a comment
! a blank line follows...

! next, a statement continuation...
PROCESSOR -
```

```
    host

! now, both features in combination...
PROCESSOR - ! comment AND continuation
    root
```

The above sequence is, to the configurer, exactly equivalent to the following:

```
PROCESSOR HOST
PROCESSOR ROOT
```

The various facilities used above can be summarised as follows:

- Case of letters is not significant to the configurer; in other words, upper and lower case letters may be used interchangeably.
- White space within a line (space characters, tab characters and so forth) is compressed; for example, three consecutive spaces would be seen as one.
- Everything from an exclamation mark character '!' to the end of the line is taken to be a comment, and is discarded.
- If the last non-whitespace character on a line is a hyphen '-', the line is taken to be continued onto the next line.
- Continuation and commenting can be used together; the hyphen must then be the last non-whitespace character before the comment.

In addition to these line formatting considerations, note that the configurer can accept any number of input files rather than simply one. This facility is designed to allow different parts of the description of an application to be held in separate files. For example, the description of the physical network might be held in one file and the description of the user's application in another. The configurer simply treats each input file in order as part of one long input stream.

26.2.2 Numeric Constants

Several different kinds of numeric constant are available to meet the different uses of constants within the configuration language. For example, a constant may be expressed in decimal notation or in hexadecimal.

A special notation is provided to extend the decimal constant with a scaling letter; this is most commonly used in specifications of memory allocation, which are conveniently specified in units of kilobytes or megabytes. The scaling letters 'K' and 'M' scale the decimal constant they follow by 1024 and 1024 x 1024 (1048576) respectively. Note that it is *not* possible to add a scaling letter to a hexadecimal constant; the configurer would interpret such a combination as the hexadecimal constant followed by a single-character word containing the scaling letter.

Although all numeric constants in the configuration language represent integer values, a representation including a decimal point can be used for input: the number is simply truncated towards zero before use. For example, 1.6 would simply represent 1. Because this truncation occurs after the scaling letter, if any, has been applied, the decimal point can be used to express fractions of the scaling value. For example, `1.6M` would represent 1677721, which is the truncated integer part of 1.6 x 1024 x 1024.

constant =
    decimal constant | *hex constant*;

hex constant =
    "`&`", *hex digits*;

hex digits =
    hex digit, { *hex digit* };

hex digit =
    digit | "`A`" | ... | "`F`";

decimal constant =
decimal digits, ["." , { decimal digit }], [scaling letter];

scaling letter =
"K" | "M";

decimal digits =
decimal digit, { decimal digit };

decimal digit =
"0" | ... | "9";

Some examples of numeric constants are given here, along with their values, expressed in decimal.

10	10
&10	16
10K	10240
10M	10485760
1.6	1
1.6k	1638

26.2.3 String Constants

The only circumstance in which a string constant is required in the configuration language is when an operating system file must be identified. Such string constants in the configuration language are simply enclosed in double quotes. No notation is available for including double quotes within the string; this is unnecessary as MS-DOS file names may not contain this character.

The trailing string quote may be omitted if the string is terminated by the end of the line.

string constant =
"", { *? any ASCII character other than newline or double quote ?* }, [""];

Some examples of valid string constants are as follows:

```
"string"
"c:\mytasks\x.b4"
"fred.b4
```

Note that the case of the characters in file names is significant, even though MS-DOS does not use this distinction. This is to help when the software is ported to other environments.

26.2.4 Identifiers

Each object in the physical transputer system (processors and wires) and in the user's application (tasks and connections) has a unique identifier. This is used by the configurer in error reports, and is also used to specify relationships between the objects. For example, a wire runs between links on two named processors.

Identifiers for objects in the configuration language are simply sequences of letters, digits and the special symbols underline '_' and dollar sign '$'. The sequence must start with a letter.

identifier =
    letter, { *identifier character* };

identifier character =
    letter | *digit* | "$" | "_";

letter =
    "A" | ... | "Z";

Some examples of valid identifiers follow. Note that the last three examples would all be treated identically by the configurer, because the case of letters is not significant.

```
proc_5
do$work
root
a_very_long_name
```

```
A_Very_Long_Name
A_VERY_LONG_NAME
```

Part of the syntax of each of the configuration language statement types which declare an object is the identifier which is to be used to refer to that object in later statements. For example, the identifier given to a processor is used again in placing tasks on that processor or in wiring the processor's links to those of other processors.

It is sometimes convenient, when an object will *not* be referred to later, to allow the configurer itself to choose an identifier for an object rather than for the user to invent meaningless identifiers for every object. The declaration statement types all allow a question mark to be used in place of an identifier.

new identifier =
    identifier | "?";

Normally, this special form of identifier is used when declaring wires and connections, as there is at present no statement type which refers back to these objects. Declarations of processors and tasks will almost always require an explicit identifier to be used, as these identifiers are used later when placing the tasks onto the network of processors.

An example of using the question mark form of identifier would be as follows:

```
wire ? host[0] root[0]
```

This statement declares a wire running from link number 0 on processor `host` to link number 0 on processor `root`. The configurer will be able to report errors concerning this wire by reference to the line number and file name of the declaration, but the user will not be able to refer to the wire again.

26.2.5 Statements

Given the definitions of such primitives as numeric constants and identifiers, the high-level syntax of the configuration language can now be presented. The combined input file consists of a number of newline-separated statements, as follows:

input file =
    { [*statement*], *newline* };

Note that the statement part of the above is optional, allowing for blank lines appearing between statements. This may come about either deliberately, perhaps to improve the readability of the input file, or because the line contained only a comment, which is of course not visible at this level.

Each statement in the input file is one of the following statement types. The different statement types are covered in the subsections which follow.

statement =
    processor statement |
    wire statement |
    task statement |
    connect statement |
    place statement |
    bind statement;

There is no restriction on the order in which statements appear in the input file, except that no object may be referred to before it has been declared.

26.2.6 PROCESSOR Statement

processor statement =
    "PROCESSOR", *new identifier*, { *processor attribute* };

processor attribute =
"TYPE", "=", *processor type* |
"BOOT", "=", *boot file specifier* |
"RAM", "=", *constant*;

processor type =
"PC";

boot file specifier =
string constant;

The PROCESSOR statement declares a physical processor. Every processor in the physical network must be declared, including the host processor from which the network is to be bootstrapped (normally an IBM PC-type machine). The configurer assumes that the processor named **host** is the host processor; thus, each configuration must contain a statement as follows:

```
processor host
```

Most processors declared in a configuration file will be declared so that user tasks can be placed on them by later statements. However, it is sometimes necessary to simply *describe* the tasks placed on a particular processor without causing them to be loaded into the processor. For example, the physical network may contain some processors which will already be executing tasks at the time the rest of the network is bootstrapped.

A trivial example of this case is the host processor itself. In the case of an IBM PC host processor, the host will usually be executing the **afserver** program when the network is loaded, simply because that is the program which loads the rest of the network. It is necessary to be able to specify the **afserver** task to the configurer so that its ports can be connected to ports in user tasks, but without forcing the configurer to attempt to bootstrap the IBM PC. Similarly, some processors in the network might be set to bootstrap from ROM rather than from link; here, too, there is a need to describe the tasks running in those processors without attempting to bootstrap them.

A processor is declared to the configurer as having already been bootstrapped by means of the TYPE attribute. For example, a physical network containing one transputer and two IBM PCs might be described as follows:

```
processor host
processor root_processor
processor other_IBM_PC type=pc
```

Note that the default for the host is that it is TYPE=PC already. The default for all other processors is to be normal, bootable, transputer processors.

Every processor is assumed to be able to support any user task placed on it by the configuration file; specifically, there is no way to ask the configurer to check the memory requirements of tasks placed on the processor against the amount of physical memory available. Similarly, although certain tasks may not be able to execute on particular types of processor (for example, a task making use of the floating point instructions found only on the T800 cannot execute on a T414), the configurer cannot check for this and the responsibility for ensuring a valid configuration is the user's.

Every processor in the network is assumed to have four Inmos links, numbered 0 to 3. These may be referred to (in the WIRE statement) by means of a link specifier construct, which consists of the processor identifier followed by the link number enclosed in square brackets:

link specifier =
    processor identifier, "[", *constant*, "]";

For example, link number 3 of the processor called `extra` would be specified as `extra[3]`.

26.2.6.1 BOOT Attribute

The BOOT attribute is used to indicate that a processor should not be loaded in the conventional manner but should be booted with the

contents of a named file.

```
PROCESSOR Edge  BOOT="sensor.b4"
PROCESSOR Gateway BOOT="anneal.app"
```

At load time a copy of the raw data in the boot file is simply sent to the processor: this can be any code suitable for booting a transputer, including an application image file generated by either the static or flood-fill configurers. In other words, a processor declared with the BOOT attribute can be thought of as the root processor of a sub-network to be booted using the named boot file.

In this way, a main statically-configured network can include static sub-networks or processor farm sub-networks "on the side". However, these sub-networks must be connected at the *edge* of the main network. There must be only one connection between a sub-network and the main network. If this restriction is not followed, the network may fail to load.

Only the root processor of the sub-network should be described in the main configuration file. If the boot file for the sub-network is a configured application, then a sub-network configuration file will have been used to create it. If the static configurer was used for the sub-network, the sub-network configuration file defines the sub-network topology; this description must be accurate, as no checking can be done during the main network configuration. The processor in the main network which has a BOOT attribute appears in the sub-network configuration file as the `host` processor.

The task in the main network which is to communicate with the root processor in the sub-network must have its ports bound to the appropriate link addresses. The programmer must use the actual hardware addresses for the links to do this. These addresses are as follows:

	Output address	Input address
Link 0	&80000000	&80000010
Link 1	&80000004	&80000014
Link 2	&80000008	&80000018
Link 3	&8000000C	&8000001C

The main task of the sub-network application should be linked with the stand-alone run-time library unless the task it will communicate with in the main network can respond to server protocol (e.g., if the main network task is a file multiplexer).

As an example of a sub-network, if the upc application described in chapter 5 were to be split into a main and sub-network using the BOOT attribute, the main network configuration file would look like this:

```
! MAINNET.CFG
! Configuration file for upper casing example
! using "boot=" to boot sub-network with upc

processor host
processor root
processor P001 BOOT="subnet.app"

wire ? host[0] root[0]    ! connect PC to transputer
wire ? root[1] P001[2]

! Task declarations
task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task driver ins=3 outs=3

! Assign software tasks to physical processors
place afserver host
place driver root
place filter root

! Set up the connections between the tasks.
connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]

connect ? filter[1] driver[1]
```

```
connect ? driver[1] filter[1]

! bind ports to link to sub-network root processor
bind input  driver[2] value =&80000014    ! I/O over
bind output driver[2] value =&80000004    ! link 1
```

The sub-network configuration file would look like this:

```
! SUBNET.CFG
! Configuration file for uppercasing example. When
! configured this application can be used to boot a
! processor sub-network with the upc program.

processor host          ! really root in main network
processor P001

wire ? host[1] P001[2]

! tasks
task driver ins=3 outs=3
task upc ins=1 outs=1 data=1k

place driver host
place upc P001

connect ? upc[0] driver[2]
connect ? driver[2] upc[0]
```

26.2.6.2 RAM Attribute

The RAM attribute overrides the default mechanism which dynamically determines the amount of memory available to a processor at boot time. The default mechanism probes memory to do this and with certain board designs this is not desirable.

When the RAM attribute is used the configurer will assume that the processor has the amount of memory specified as the parameter to the RAM attribute and the dynamic method of memory determination will not be used. For this reason, care should be taken to ensure that the processor really does have the amount of memory specified with the RAM attribute.

The following RAM attributes declare that processor **pe1** has 4MB of memory and processor **pe2** has only 500KB of memory.

```
processor pe1 ram=4096K
processor pe2 ram=500K
```

Use of the RAM attribute may affect the size of the application file as it may cause extra loading software to be included.

26.2.7 WIRE Statement

wire statement =
"**WIRE**", *new identifier, link specifier, link specifier;*

The WIRE statement declares a physical wire connecting links on two physical processors. Each wire supports two connections, one in either direction. The two link specifiers in the WIRE statement may therefore be interchanged without affecting the statement's meaning. For example, the following statements both declare a wire named **yellow_wire** running between link 2 of processor **proc_one** and link 3 of processor **proc_two**:

```
wire yellow_wire proc_one[2] proc_two[3]
wire yellow_wire proc_two[3] proc_one[2]
```

Although it is only necessary to declare the wires which are actually used by the application, in practice it is advisable to declare all the wires. This is because the configurer may be able to use the extra wires for booting the application, and as a result may be able to reduce the size of the boot file by eliminating some of the loading software.

26.2.8 TASK Statement

task statement =
"**TASK**", *new identifier,* { *task attribute* };

task attribute =
"INS", "=", *constant* |
"OUTS", "=", *constant* |
"FILE", "=", *task file specifier* |
"OPT", "=", *opt area* |
"URGENT" |
memory area, "=", *memory amount*;

opt area =
memory area | "CODE";

memory area =
"STACK" | "HEAP" | "STATIC" | "DATA";

memory amount =
constant | "?";

task file specifier =
identifier | *string constant*;

The TASK statement declares a task, which may be either a user-supplied task or one of the standard tasks provided with the configurer. Each task statement may contain a number of task attribute clauses, each of which describes some aspect of the task. The task's attributes may appear in any order within the statement.

26.2.8.1 INS Attribute

Each task declaration must include an INS attribute, which specifies the number of elements in the task's vector of input ports. If the task needs no input ports (because it only requires to send messages to other tasks, never to receive) then the number of input ports may be specified as 0.

26.2.8.2 OUTS Attribute

Each task declaration must include an OUTS attribute, which specifies the number of elements in the task's vector of output ports. If the task needs no output ports (because it only requires to receive messages from other tasks, never to send) then the number of output ports may be specified as 0.

26.2.8.3 FILE Attribute

This attribute specifies the file in which the memory image of the task is to be found. Task image files are produced by the linker program `linkt`.

The FILE attribute is ignored for any processor which is declared as already having been bootstrapped, and may be omitted. This state is assumed for the host processor and for any processor for which the processor attribute `type=pc` has been specified.

If the FILE attribute is omitted for a normal (bootable) processor, the configurer will scan the current directory and the directories specified in the MS-DOS environmental variable `PATH` for a file whose name is the same as the task's name, with the suffix "`.b4`". The search stops at the first directory in which a file with the appropriate name is found. For example, take the TASK statement `TASK THIS` with no FILE attribute, with the MS-DOS `PATH` variable set up as follows:

```
PATH=c:\mytasks;c:\dos;c:\tputer
```

In this case, the configurer would search for the task image in the following files, in order:

```
.\this.b4
c:\mytasks\this.b4
c:\dos\this.b4
c:\tputer\this.b4
```

If the FILE attribute is present, its argument is either a string constant, or a word with the same syntax as an identifier. In the former case, the string is the name of the file which will be opened, as in the following example:

```
task x file="c:\mytasks\mytask.b4" ...
```

If the identifier-like option is taken, the identifier given is used in a search through the MS-DOS **PATH** in the same way as the task's own identifier would have been if the FILE attribute had been omitted:

```
task x file=mytask ...
```

26.2.8.4 Memory Size Attributes

The various memory size attributes specify the size of the various areas used as workspace for the task, as well as specifying which memory allocation strategy should be used.

The argument to one of the memory size attributes is an integer expressing the number of bytes of memory to be allocated to the area in question. Sizes smaller than 128 bytes will not be accepted, to prevent accidental entry of unreasonably small amounts (for example, by typing `1.6` instead of `1.6K`). It is also possible to specify "the rest of memory available on the processor" by entering a question mark instead of an integer. Only one task may request this treatment on any particular processor.

The *single-vector* allocation strategy is used if the DATA attribute appears. In this strategy, the task uses a single area of memory for all workspace requirements, whether stack, heap or static data. The stack and heap are allocated at opposite ends of this area, and grow towards each other. For example:

```
task x ... data=50k ...
```

The *double-vector* allocation strategy is used if the STACK and HEAP attributes appear (STATIC is available as a synonym for

HEAP). In this strategy, the stack occupies a separate area of memory to all the other workspace used by a task. This can be useful when a task has a small stack requirement, as it can allow for the stack area to be placed into the transputer's on-chip memory using the task OPT attribute; this technique can produce large performance benefits. An example of double-vector allocation is as follows:

```
task x ... stack=1k heap=10k ...
```

The two allocation strategies are mutually exclusive. Thus, if the DATA size for the task is given, neither STACK nor HEAP should appear. If the two-vector allocation strategy is chosen, both STACK and HEAP must be specified. If no memory size attributes at all appear for a task, the default is the same as `DATA=?`; in other words, single-vector allocation of the rest of memory available on the processor.

26.2.8.5 OPT Attribute

This attribute specifies that the memory area given as its argument should be placed, if possible, into the transputer's on-chip memory area. The CODE specifier indicates the area of memory which will contain the executing code of the task; the other memory area specifiers have the same interpretation as for the memory size attributes.

If not all of the memory areas specified will fit into the on-chip memory, then some will be placed instead into the slower external memory, which is the default allocation for all memory areas. The order of precedence between memory areas in the same task is: stack, code, heap. In other words, if `OPT=STACK` and `OPT=CODE` are both specified, then the stack area is more likely to be placed in on-chip memory. No order of precedence is guaranteed between memory areas in different tasks.

It is possible for only part of a memory area to be placed in the on-chip RAM; this is useful in respect of the code area, where the modules which appeared first in the linker command line will have

been placed at the start of the code area. If the most critical procedures are placed in the first module, then the likelihood of their being executed from on-chip memory will be increased.

The on-chip memory is quite small (2KB on the T414, 4KB on the T800), so the OPT attribute should be used sparingly to ensure that critical memory areas are not displaced into the slower external memory by less critical memory areas.

An example of a critical task with small stack and large data requirements might be as follows:

```
task t stack=1k heap=100k -
  opt=stack opt=code
```

26.2.8.6 URGENT Attribute

This attribute specifies that the task's initial thread is to be started at the urgent priority level. The default is that the task's initial thread is started at the not-urgent priority level. For example:

```
task x ... urgent ...
```

26.2.8.7 Port Specifiers

After the declaration of a task, its ports may be referred to in much the same way as the links of a processor, by a port specifier construct consisting of the task identifier followed by a number enclosed in square brackets:

port specifier =
    task identifier, "[", *constant*, "]";

For example, either input or output port number 5 on task **user** would be specified as **user[5]**.

Note that a port specifier as given here does not indicate whether the port concerned is an input port or an output port, that is, whether

the index given is into the task's vector of input ports or into its vector of output ports. This information is provided by the context in which the port specifier appears. In the CONNECT statement, the port specifier's direction is determined by its position within the line. In the BIND statement, the port specifier is preceded by a direction word (INPUT or OUTPUT).

26.2.9 CONNECT Statement

connect statement =
"CONNECT", *new identifier, output port specifier,*
input port specifier;

output port specifier =
port specifier;

input port specifier =
port specifier;

The CONNECT statement connects an output port on one task with an input port on another task. For example:

```
connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]
```

Note that the order of the ports given in the CONNECT statement is significant, unlike the order of the links in the WIRE statement which CONNECT otherwise resembles.

26.2.10 PLACE Statement

place statement =
"PLACE", *task identifier, processor identifier;*

processor identifier =
identifier;

task identifier =
    identifier;

The PLACE statement determines which processor a particular task is to execute on; every task must be placed on some processor. A simple example of the use of this statement might be as follows:

```
place user_task root
place afserver host
```

Where multiple tasks which have the same image file are placed on the same processor, they all share a single instance of the image code. This helps to save space and can be particularly useful for the simulation of large regular systems on fewer processors than will eventually be used.

Note that it is incorrect to PLACE a task on a processor which was declared with a BOOT attribute or on any processor which can only be reached from the host via processors declared with the BOOT attribute.

26.2.11 BIND Statement

bind statement =
    "BIND", *binding type, port specifier, binding value;*

binding type =
    "INPUT" | "OUTPUT";

binding value =
    "VALUE", "=", *constant;*

The BIND statement allows the contents of a port to be explicitly set to some literal value. Normally, ports are only bound by means of the CONNECT statement; ports left unbound are pointed at unique transputer channel words so that attempts to send or receive messages through them cause the minimum of harm; the thread causing the attempt to communicate over the unbound port simply

pauses indefinitely rather than causing failure of possibly all threads running on the processor.

One application of the BIND statement is to give a task access to the transputer's external event mechanism. This appears as a channel word at address 80000020_{16}. Input port 5 of task **event_handler** could be initialised to point to this channel word as follows:

```
bind input event_handler[5] value=&80000020
```

Another application of the BIND statement is to pass an integer parameter to a user task. Here, the same input port as before is bound to the value 5:

```
bind input event_handler[5] value=5
```

This technique can be used to allow several otherwise identical tasks to behave differently. For example, tasks executing on a fast processor can have this fact indicated to them by means of a parameter value, and use a more processing-intensive algorithm for the solution of some problem. Another use of this parameter facility is to "label" each task with a unique identifier.

Note that if an arbitrary value is supplied for a port binding and an attempt is then made to send or receive a message using that port, the processor on which the task resides will most probably crash.

Chapter 27

Flood-Fill Configurer Reference

There are two types of user task in a flood-fill configured application. One task, referred to as the *master*, divides up the computation to be performed into small *work packets*. The other task, which is known as the *worker*, is replicated all over the network; it accepts work packets originating from the master, performs some computation and sends a reply packet or packets back.

27.1 User Task Protocol

This section describes the protocol used by the user tasks in a flood-filled application. Note that a different protocol may well be used by the router tasks, for example to avoid problems with T414A restrictions on minimum length of messages sent across links.

27.1.1 Master Task's Ports

The master task has two input ports and two output ports. The input and output ports `master[1]` are connected in the usual way to a file server task such as `afserver` (probably via a protocol filter task such as `filter`).

The input and output ports `master[0]` are connected to the *router* task. The router task is provided by the flood-fill configurer, and has the function of transporting work packets from the master through the network to idle workers to be processed.

27.1.2 Worker Task's Ports

Each worker task has one input port and one output port. These ports `worker[0]` are connected to the part of the routing system which exists on each processing node of the network.

27.2 Packet Format

Work and response packets have identical format, consisting of a fixed-length portion and an optional variable-length portion. The two portions of the packet are send as separate messages. Each packet starts with a message containing a 4-byte integer header, as shown in Figure 27.1.

The various fields of this 32-bit message are used as follows:

- The least-significant sixteen bits of the message are used to indicate the length of the data block following the header. If the length is zero, no data block follows; otherwise this many bytes of additional data follow as a separate message of that length.
- Bit number 16 (value 00010000_{16}) is always 1.

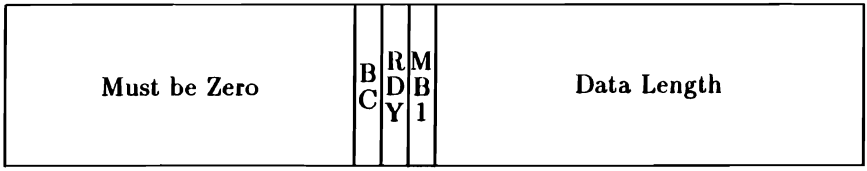


Figure 27.1: Format of Packet Header

- Bit number 17 (value 00020000_{16}) is set to 1 to signify that the sending task is *ready*. A worker task can set RDY = 0 to indicate that further response packets will be issued before the next work packet will be accepted.
- Bit number 18 (value 00040000_{16}) is set to 1 to signify that this packet is a broadcast.
- Bits number 19-31 are always 0.

Chapter 28

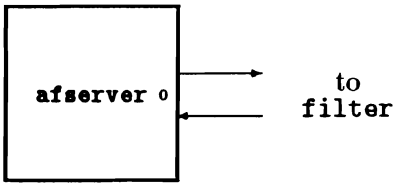
Task Data Sheets

This chapter contains descriptions of the standard "building block" tasks which are provided with Parallel Fortran.

The description of each task starts with a diagram indicating the way in which the ports of the task should be connected to those of other tasks. Small digits inside the box representing the task are used to indicate port numbers corresponding to the connections visible outside the box.

This diagrammatic description is then backed up by a detailed description of the function of the task, along with examples of how a reference to the task might appear in a configuration file.

Data Sheet: afserver



The **afserver** task is used in configured applications to represent an **afserver** program executing on the host computer. It is therefore not provided in true task-image form.

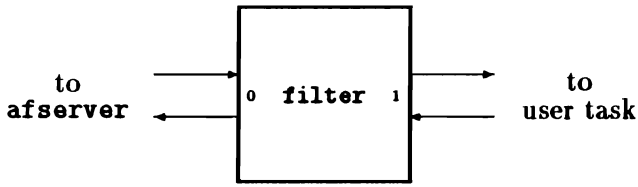
The **afserver** task should be described to the configurer as follows:

```
task afserver ins=1 outs=1
place afserver host
```

The **afserver** program (and therefore the **afserver** task) provides access to the host computer for tasks running in the transputer system, with which it communicates over its port pair 0.

The protocol used by the **afserver** is a special variant of the Inmos tagged file-server protocol, adjusted to be tolerant of a problem in the T414A which prevents one-byte messages being sent over links. The **afserver** would therefore normally be attached to a **filter** task so that this variant protocol could be converted back into the protocol which is used by user tasks.

Data Sheet: `filter`



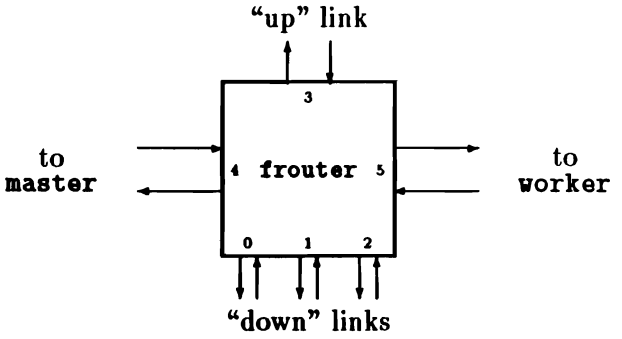
The **filter** task is used to convert between the two extant variants of the Inmos tagged file-server protocol. The two variants arise because of a problem with T414A transputers, which cannot send one-byte messages across links. A **filter** task would be described in a configuration file as follows:

```
task filter ins=2 outs=2 data=10k
```

A **filter** task's port pair 0 communicates using the T414A-tolerant variant of the Inmos protocol. This is normally attached to an **afserver** task running on the host computer. Port pair 1 of a **filter** task communicates using the standard version of the Inmos protocol.

Thus, if a **filter** task is interposed between an **afserver** and a user task, they will be able to communicate normally although each is using a different protocol.

Data Sheet: `frouter`



The `frouter` task is used by the flood-filling configurer as the standard task which resides on each node of a flood-filled network and manages the flow of work packets and responses through the network.

The attributes used by the flood-filling configurer for the `frouter` task are as follows:

```
task router file=frouter ins=6 outs=6 -
            data=11k urgent
```

The following list summarises the way in which the `frouter` task is used by the flood-filling configurer:

0–2 Each of these pairs of "down" ports are either set to zero by the loader, or are connected to the "up" ports of nodes deeper in the network which were bootstrapped from this node. For each non-zero port pair in this range, the `frouter` task will start a pair of threads to carry packets to and from the subnetwork attached through that link.

3 If this node is *not* the root of the network, these "up" ports are connected to a pair of "down" ports of the router on the

node which bootstrapped this node. In this case, the `frouter` task will read work packets and send responses to the booting node (and thus ultimately to the master task executing on the root node) through this pair of ports. If this node is the root of the network, these ports are set to zero by the loader and are ignored by the `frouter` task: port pair 4 (attached to the master task) will be used instead.

4 If this node is the root of the network, these ports are connected to the master task. In this case, the `frouter` task will read work packets and send responses to the master task through this port pair. Otherwise, these ports are set to zero by the loader and the `frouter` task will use port pair 3 to reach the master task.

5 These ports are connected to the worker task executing on this node.

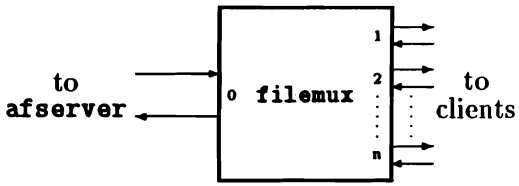
The standard `frouter` task uses two protocols in communicating with the tasks to which it is connected:

4–5 Port pairs connected directly to user tasks use the standard "net" protocol described in section 27.1.

0–3 Port pairs connected to other routers through Inmos links use a variant of the "net" protocol which is tolerant to the T414A problem with one-byte messages. In this variant, a two-byte message is actually transferred whenever the message header indicates that a one-byte message should follow.

Note that a communications task like `frouter` should normally be specified as having the URGENT attribute. This prevents worker tasks in the network becoming idle because there is too little CPU time available elsewhere in the network for the router to operate.

Data Sheet: filemux



The **filemux** task allows several client tasks to share a single file server task by merging (multiplexing) the clients' request streams into a single stream of requests. This allows more than one task in a Parallel Fortran application to use standard file I/O.

In a simple system, the "to **afserver**" ports are connected to the **afserver** via a **filter** task. However, they may be connected to any task which accepts the **afserver** protocol. In particular, they may be connected up as the client of another **filemux** task to build multiplexer chains.

In general, **filemux** simply passes on service requests from its clients and forwards the responses. The exception is the "server terminate" request. The multiplexer will only pass on "server terminate" once all its clients have requested server termination.

Figure 28.1 shows the basic problem with which the multiplexer task is intended to assist. Here, the task **server** runs on the host and provides file services via a protocol filter task **filter** to a client task **client_1**. The **filter**, **client_1** and **client_2** tasks all run in the transputer system. The difficulty is in arranging that the second client task **client_2** can gain access to files stored on the host processor.

One possibility is to connect the two client tasks together and arrange for **client_2** to request file services from **client_1**. An-

other possibility, illustrated in figure 28.2, is to introduce a new task **multiplexer** designed to solve this particular problem. The multiplexer task is connected to both client tasks and passes their file service requests through to the filter and thus the server on the host system.

Although it is possible to build any required multiplexing system by combinations of the $2 \rightarrow 1$ multiplexer shown in figure 28.2, the **filemux** task is more general in that it can handle any number of client tasks: it performs an $n \rightarrow 1$ multiplexer function. Port pair 0 (i.e., input port 0 and output port 0) of the multiplexer is always connected to the task from which file services may be obtained; in this example, the filter task. All other port pairs supplied to the multiplexer in configuration language statements like **ins=***n*, **outs=***n* are connected to a total of $n - 1$ client tasks.

In this way, the multiplexer can handle any number of client tasks, as long as the user provides it with sufficient memory to support them. The multiplexer needs no more than $(6 + 0.25n)$K bytes of data storage, where n is the number of tasks supported. The user should use a DATA attribute to the multiplexer's TASK statement to ensure this memory is available.

An example of a configuration file which represents the configuration of tasks shown in figure 28.2 is given in figure 28.3 (the PROCESSOR and PLACE statements required have been omitted for clarity.)

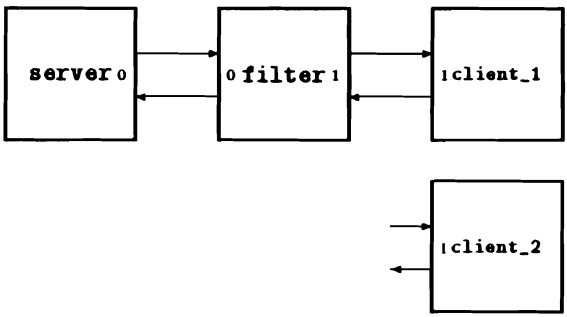


Figure 28.1: Limitation on Server Connections

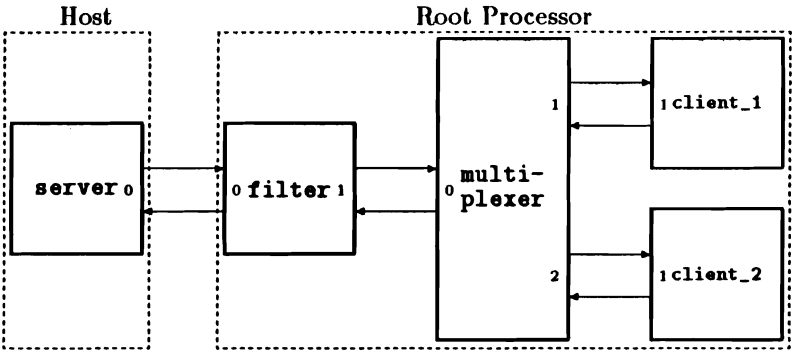


Figure 28.2: Using the Multiplexer

The multiplexer task may also be used to support client tasks which are not running in the root processor. When they are running in an adjacent processor and there is a spare **wire** connecting the two processors, as in figure 28.4, then no additional work needs to be done; the configurer will simply run the connection between the client and the multiplexer across any available wire. Note that each **wire** between processors, defined in the configuration file, supports bi-directional communication between two tasks, one on each processor.

However, if the client task is some distance away, the multiplexer can be used in a $1 \rightarrow 1$ configuration (i.e., serving only one client) to pass file service requests through processors in the middle of the network until finally reaching the multiplexer in the root processor, which is connected to the filter task and thus the server as shown in figure 28.5. Thus, a network of transputers might contain a tree of multiplexer tasks, each passing file service requests up towards the root. This kind of arrangement can be continued indefinitely as long as the server task has sufficient resources to handle all the clients together.

As mentioned earlier, the multiplexer can be used in an $n \rightarrow 1$ manner. An example of its use with eight client tasks (i.e an $8 \rightarrow 1$ multiplexer) is shown in figure 28.6. It should be noted that the

```
task server ins=1 outs=1

task filter ins=2 outs=2 data=10k
connect ? filter[0] server[0]
connect ? server[0] filter[0]

task multiplexer file=filemux ins=3 outs=3 data=10k
connect ? filter[1] multiplexer[0]
connect ? multiplexer[0] filter[1]

task client_1 ins=2 outs=2 data=50k

connect ? multiplexer[1] client_1[1]
connect ? client_1[1] multiplexer[1]

task client_2 ins=2 outs=2 data=50k
connect ? multiplexer[2] client_2[1]
connect ? client_2[1] multiplexer[2]
```

Figure 28.3: Example Configuration File

multiplexer port pair 0 may be connected to one of the client port pairs of another multiplexer task. This allows multiplexers to be chained together to provide file services across a network, if there are sufficient links available to do this. Similarly, the `client_8` task might itself be a multiplexer providing file services to tasks on an adjacent processor.

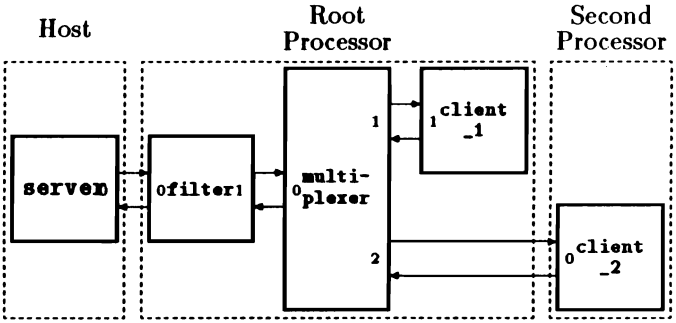


Figure 28.4: Using the Multiplexer on an Adjacent Processor

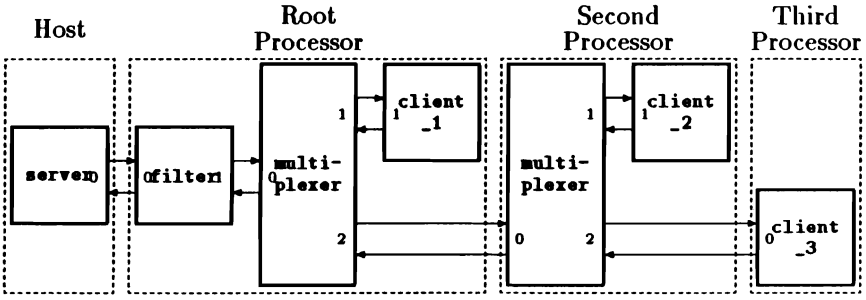


Figure 28.5: Using the Multiplexer from Within a Network

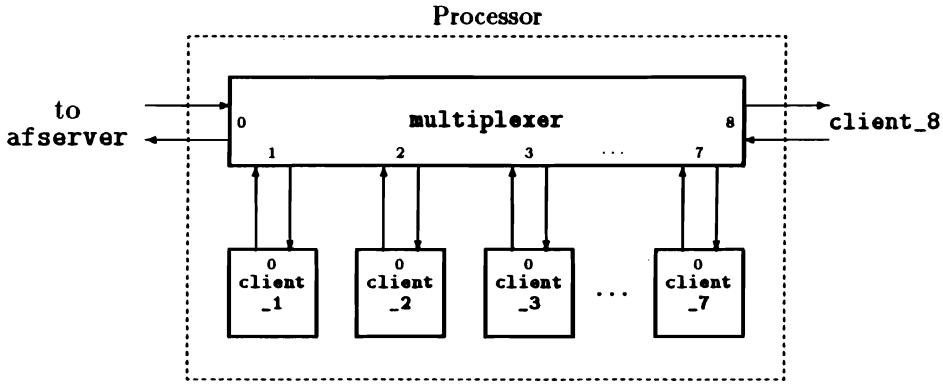
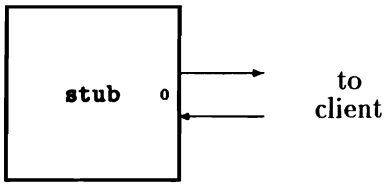


Figure 28.6: Using an 8 → 1 Multiplexer

Data Sheet: stub



Tasks which are not connected to the **afserver** or a file multiplexer task are normally linked with the standalone C run-time library. There are some Fortran facilities, such as internal file I/O, ENCODE and DECODE which do not strictly require file server support but which cannot be supported by the standalone run-time library. The **stub** filer task allows you to write standalone tasks which make use of such facilities.

All Fortran I/O facilities which do not actually require **afserver** support can be made available to a standalone task by linking it with the full standard run-time library. If facilities such as non-internal file I/O which require server support are not used, the standard library will only attempt to communicate with the server when it tries to read command line arguments at program startup and set the exit status at shutdown. The **stub** filer task acts as a sink for these communications: it accepts this limited subset of the **afserver** protocol from its client task and sends back stylised dummy replies.

Note that if the **stub** filer's client task does try to use some facility which requires server support, the **stub** filer will either send back a meaningless response, or terminate and leave the **client** task deadlocked waiting for a response to its request.

The **stub** filer task is connected to its client as shown in the example below. The run-time library always uses output port 1 and input

port 1 to communicate with the server, so the client's port pair 1 must be connected to the **stub** filer's port pair 0.

```
task stub ins=1 outs=1 data=20k
task client ins=3 outs=3

connect ? client[1] stub[0]
connect ? stub[0] client[1]
```

The **stub** filer and its client task act together like an ordinary standalone task. In the example above the **client** task has been given three input and three output ports. Port pairs 0 and 1 are reserved for use by the run-time library, so port pair 2 is left free for communication with other tasks.

Use the standalone run-time library in preference to the **stub** filer if possible. It is simpler, and the memory used for the **stub** task and some of the startup and shutdown overhead in the full run-time library is saved.

The **stub** filer can only be used with the static configurer, **config**; it cannot be used with the worker task of a flood-filled application. The flood configurer, **fconfig**, will not allow you to specify that a task other than the worker is to be replicated throughout the network.

Appendix A

Distribution Kit

This appendix lists the files which make up the distribution kit for this version of Parallel Fortran. Each file name is accompanied by a short description of the file's function.

A.1 Directory \tf2v1

`afserver.exe`	file server for IBM-PC and compatible hosts
`necserve.exe`	file server for NEC PC-9801 hosts
`t4f.exe`	Fortran compiler driver program for T4
`t8f.exe`	Fortran compiler driver program for T8
`tf.exe`	generic Fortran compiler driver program
`tf.b4`	Fortran compiler code for T4 and T8
`alt.inc`	run-time library package files
`chan.inc`	
`dos.inc`	
`misc.inc`	
`net.inc`	
`sema.inc`	
`thread.inc`	
`timer.inc`	

`linkt.b4`	linker code
`linkt.exe`	linker driver program
`t4flink.bat`	batch file to invoke linker for T4
`t8flink.bat`	batch file to invoke linker for T8
`t4ftask.bat`	batch file to link a task for T4
`t4fstask.bat`	batch file to link a stand-alone task for T4
`t8ftask.bat`	batch file to link a task for T8
`t8fstask.bat`	batch file to link a stand-alone task for T8
`frtlt4.bin`	Fortran run-time library for T4 only
`frtlt8.bin`	Fortran run-time library for T8 only
`safrtlt4.bin`	standalone Fortran run-time library for T4
`safrtlt8.bin`	standalone Fortran run-time library for T8
`t4harn.bin`	T4 harness code
`t8harn.bin`	T8 harness code
`taskharn.t4`	harness for tasks on T4
`taskharn.t8`	harness for tasks on T8
`config.b4`	configurer code
`config.exe`	configurer driver program
`fconfig.b4`	flood configurer
`fconfig.exe`	flood configurer driver program
`floader.b4`	loader code used by `fconfig`
`gloader.b4`	loader code used by `config`
`decode.b4`	`decode` utility code
`decode.exe`	`decode` utility driver program
`fpr.b4`	`fpr` utility code
`fpr.exe`	`fpr` driver program
`mempatch.b4`	`mempatch` utility code
`mempatch.exe`	`mempatch` utility driver program
`tnm.b4`	`tnm` utility code
`tnm.exe`	`tnm` utility driver program
`tunlib.b4`	`tunlib` utility code
`tunlib.exe`	`tunlib` utility driver program
`worm.b4`	`worm` utility code
`worm.exe`	`worm` utility driver program

`filter.b4`	afserver protocol filter task
`filemux.b4`	file service multiplexer task
`frouter.b4`	standard flood router task
`stub.b4`	stub filer task

A.2 Directory \tf2v1\examples

`hello.f77`	"Hello, world!" program
`cga.f77`	source package of functions to access PC's CGA display hardware from the transputer. Provides an example of use of DOS-access functions.
`cga.inc`	header file for the above
`mandelm.f77`	source of "master" part of Mandelbrot example
`mandelw.f77`	source of "worker" part of Mandelbrot example
`mandelm.lnk`	module list for linking Mandelbrot master
`mandelw.lnk`	module list for linking Mandelbrot worker
`command.inc`	command packet format for Mandelbrot example
`results.inc`	results packet format for Mnadelbrot example
`mandel.cfg`	configuration file for Mandelbrot example
`fmandel.cfg`	configuration file for flood-filled version of Mandelbrot example
`mandel.bat`	batch file to compile, link and configure Mandelbrot example
`driver.f77`	source of upper-case I/O task
`upc.f77`	source of upper-case conversion task
`upc.cfg`	configuration file for upper-case example

Appendix B

Compatibility with T414A and T800A

This appendix describes the problems which you may encounter if you run Parallel Fortran programs on early transputer chips.

We recommend that if you have one of the development systems sold with these early pre-production processors, you should have it upgraded with a production processor. Failing this, the various problem areas are listed here so that you can program round them.

B.1 Problems with T414A

Note that the pre-production T414 (mask revision A) *cannot* simply be replaced by a later revision T414 without making changes to the support circuitry. This is because various details of the external clock and phase-locked-loop circuitry differ between the T414A and all later transputer processors. For their own B004 board, Inmos can provide an upgrade kit (IMS B901) which includes a T414B chip, an extraction/insertion tool and full instructions on the modifications required.

B.1.1 Restriction on Message Lengths

The T414A cannot reliably transmit a single-byte message across a link. Message transfer across internal channels is not affected.

This problem should not affect users of single-transputer systems, as the `filter` task used to communicate with the `afserver` task takes care of this problem. Similarly, the private protocol used between routers in a flood-filled network avoids this problem by padding out 1-byte messages to two bytes for transmission. User tasks in both of these cases are unaware of the protocol conversions.

This problem can be easily avoided in new systems by ensuring that protocols never include single-byte messages.

B.1.2 Problems with Timers

B.1.2.1 Timer Rate Problem

In production transputers, the timer associated with high-priority ("urgent") threads ticks once every 1μS, while the low-priority timer (that associated with "not urgent" threads) ticks once every 64μS. In the T414A, both timers tick every 1.6μS.

This problem will affect the subprograms of the `TIMER` package, and those functions in the `CHAN` package whose names end with `_T`.

B.1.2.2 Short Delay Problem

The T414A cannot reliably delay for small amounts of time (below about 5 ticks). When such an operation is attempted, the thread requesting the operation may hang forever.

This problem affects the `F77_TIMER_WAIT` and `F77_TIMER_DELAY` subroutines when small delays are specified, and the `F77_THREAD_DESCHEDULE` subroutine, which is equivalent to a 1-tick delay.

B.1.2.3 Only One Delaying Thread Problem

Only one thread on a T414A processor can be delaying at any one time. This problem will affect the `F77_TIMER_DELAY` and `F77_TIMER_WAIT` subroutines, and those functions in the `CHAN` package whose names end with `_T`.

B.2 Problems with T800A

B.2.1 Floating-Point Conversion Problems

The T800A has a problem in its floating-point microcode; the wrong result may be obtained for expressions containing integer to floating-point conversions.

The Parallel Fortran compiler has an option switch to avoid such instruction sequences; refer to section 17.2.4 for details of the `/T8A` option.

Note that the run-time library supplied with Parallel Fortran has been compiled with this option and can therefore be used safely on a T800A.

B.2.2 Instruction Decode Problems

The T800A decodes the `move2dzero` and `move2dnonzero` instructions wrongly, with the effect that when one is requested, the other is executed. Later T800 processors decode these instructions correctly, however.

Note that the `/T8A` compiler option does *not* change the behaviour of the assembler with respect to these instructions. The compiler always generates the code for the instruction as written.

Appendix C

Building a Network

In order to make use of the multi-processor facilities provided by Parallel Fortran, it is of course necessary to build a multi-transputer network on which to run the programs. This appendix describes the principles involved, and shows how to build such a network out of plug-in transputer development cards for the IBM PC.

C.1 Network Principles

There are two sets of connections to make when building a network of transputer processors. The most obvious of these are the link connecting one transputer to another; it is through these wires that the tasks running on each processor communicate with their neighbours, and through which the network is bootstrapped. An application running on a transputer network is usually aware of the topology of link connections.

Less obviously, another set of connections must be made in order to arrange that various *system services* are available to the network. Specifically, each transputer processor has *reset* and *analyse* inputs

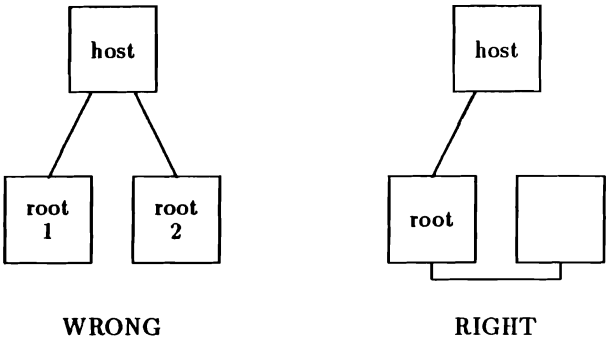


Figure C.1: "One Root" Condition

and an *error* output. The topology of the system service connections need not be related to that of the link connections.

C.2 Network Requirements

C.2.1 Requirements for Links

When building a network, there are two conditions which the arrangement of link connections must satisfy:

- Exactly one processor must be connected to the host processor. The former is referred to as the *root* processor, because it forms the root of the tree of processors in the network. Figure C.1 shows two networks, one of which is not acceptable because it attempts to have two root processors.

- each processor in the network must be reachable by a series of "hops" through links, starting at the host processor. In other words, the network must be *connected;* i.e., have no isolated nodes. Figure C.2 shows two networks, one of which is not acceptable because it has isolated processors.

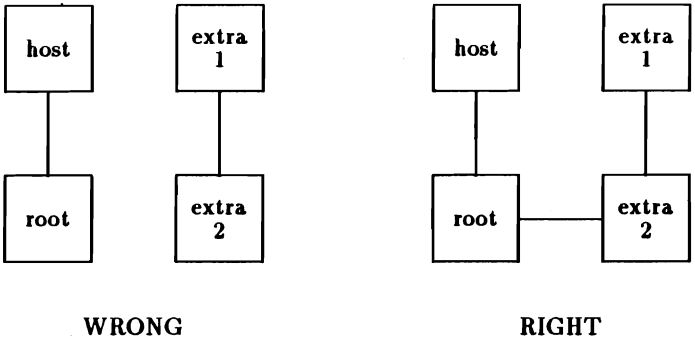


Figure C.2: "Connected" Condition

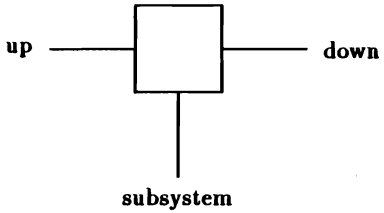


Figure C.3: Inmos System Services Scheme

C.2.2 Requirements for System Services

The only requirement which Parallel Fortran places on the arrangement of system service connections is that, immediately prior to a network being bootstrapped, all of the processors in that network must have been *reset*. Parallel Fortran makes no use of the transputer analyse and error signals at present.

The reset signal may be carried to each of the processors in the network in many different ways. However, one popular scheme is shown in figure C.3. In this scheme, each processor has three connectors:

- **UP** leads to a processor closer to the host.
- **DOWN** leads to a processor further from the host.

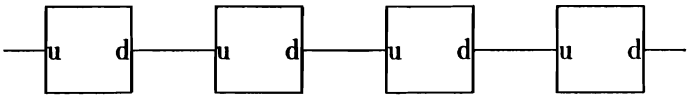


Figure C.4: System Service Daisy Chain

- **SUBSYSTEM** leads to a sub-tree of processors under the control of this one.

The system service signals are carried through from "up" to "down" so that several processors can be "daisy-chained" together. The unconnected "up" port of such a chain can be used to control the entire chain, as shown in figure C.4.

The purpose of the "subsystem" connector is to allow one processor to control others; system service signals are sometimes, but not always, also carried through to the "subsystem" connector.

C.3 Connecting a Network

This section describes how to connect up a network using boards compatable with the Inmos IMS B004 development board for the IBM PC. The B004 board is shown in figure C.5.

At the far right-hand side of this board, visible from the back of the PC in which the board has been installed, are an array of connectors by which the board may be connected to other boards. There are two columns of five connectors in this array, defined as follows:

PC link	unused
Link 0	Link 1
Link 2	Link 3
PC Reset	Subsystem
Up	Down

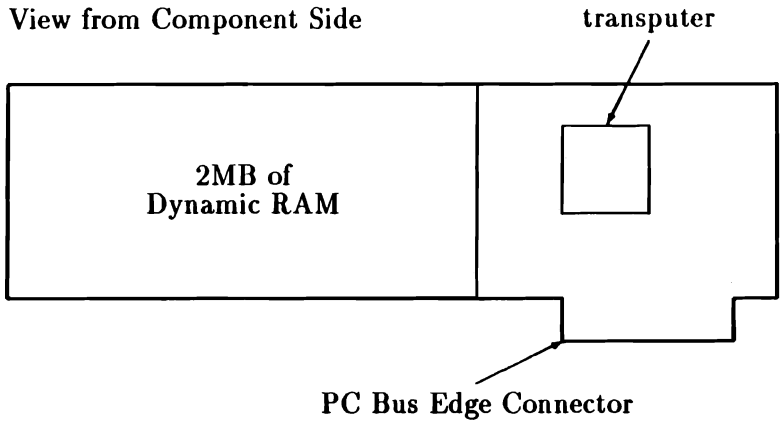


Figure C.5: B004-type Single-transputer Development Board

Boards are supplied with two "jumper" plugs and three cables. Each of these objects are arranged so that they can only fit into the connectors for which they are intended.

When only one development board is in use, the two jumpers are installed. These connect "PC Link" to "Link 0" and "PC Reset" to "Up"; in other words, the board will be reset by the PC in which it is installed, which will load it through its link 0.

To extend this basic configuration with another processor, the second board could be placed in an adjacent PC bus slot (normally to the right) and connections made to carry system services and application messages. For example, link 1 on the root transputer (the original one) could be connected to link 0 on the second board, and "Down" on the root could be connected to "Up" on the add-on. If the two boards are in adjacent slots in the PC card cage, these connectors will be adjacent as well.

This scheme can be extended to any number of development boards; the root (placed on the left) is controlled by the PC, while each board other than the right-most passes the system service signals on to the one on its right.

Appendix D

Additional Language Features

Parallel Fortran includes various features which are provided to help programmers when porting programs from certain older compilers. These facilities are:

- The `ENCODE` and `DECODE` statements;
- The `DEFINE FILE` statement;
- Another method for selecting records when using a direct access file;
- The `FIND` statement.

None of these facilities is specified by the ANSI standard, and the compiler supports all the corresponding standard methods for doing these things. If possible the standard methods should be used when writing new Parallel Fortran programs, as this will help to avoid problems when porting to other Fortran 77 compilers.

D.1 The ENCODE and DECODE statements

These statements are an alternative method for doing internal I/O. Similar results can be obtained by using READ and WRITE on internal files, as described in section 16.7.

ENCODE converts a list of variables into external format in memory, while DECODE converts a string in external format into internal values and stores them in a list of variables. They have the following formats:

```
ENCODE (count,format,buffer,IOSTAT=ios,ERR=err) list
DECODE (count,format,buffer,IOSTAT=ios,ERR=err) list
```

where:

buffer is a variable or array which contains, or will contain, the character string.

count is an integer expression which specifies the length of the buffer. In the ENCODE statement *count* is the number of characters which will be generated; if necessary, extra space characters will be generated to fill the buffer. In the DECODE statement *count* is the number of characters to be converted to internal form.

format is a format identifier. '*' is not accepted. If more than one record is specified in the format, an error occurs. The interaction between the format and the I/O list is the same as for formatted READ and WRITE statements, and is described in section 16.2.2.

ios is an integer variable where an I/O status code is placed. See section 16.3.1.1.

err is a label, to which control is transferred if an error happens. See section 16.3.1.1.

list is an I/O list. In the ENCODE statement, *list* contains the data to be converted to character form. In the

DECODE statement, *list* receives the data after conversion to internal form.

The IOSTAT and ERR clauses, with their preceding commas, may be omitted.

ENCODE works like a WRITE statement, converting the list items under the control of the format specifier and storing the generated characters in *buffer*.

DECODE works like a READ statement, converting the characters in *buffer* under the control of the format specifier and storing the internal values in the list items.

D.2 The DEFINE FILE statement

This statement specifies the characteristics of an unformatted direct access file and associates it with a I/O unit number. The OPEN statement performs a similar function (see section 16.8.2), and its use is preferred.

The DEFINE FILE statement has the following form.

```
DEFINE FILE u(rn,rl,U,asv)
```

where:

u is an integer constant or variable which specifies I/O unit to which the file is to be connected.

rn is an integer constant or variable which specifies the total number of records in the file.

rl an integer constant or variable which specifies the record length, in units of two bytes.

U is a compulsory item, specifying an unformatted file, the only type allowed.

asv is the *associated variable*, an integer variable which, after every direct access I/O operation, is set to the record number of the next record. The associated variable may not be a dummy argument.

More than one file may be specified with one DEFINE FILE statement:

```
DEFINE FILE u(rn,rl,U,asv), u(rn,rl,U,asv)...
```

The DEFINE FILE statement specifies that the file connected to I/O unit *u* contains *rn* fixed-length records, each $2 \times rl$ bytes long. The records in the file are numbered sequentially from 1 to *rn*.

The DEFINE FILE statement does not itself open the file. This is done when the first I/O statement on the specified unit is executed. If this statement is a WRITE, a new direct access file is created. If it is a READ or FIND, the file is assumed to exist already, and an error occurs if it does not. The DEFINE FILE statement must be executed before the first I/O statement on the unit.

The DEFINE FILE statement does not allow the programmer to specify a filename. Instead, a default filename is used, as discussed in section 16.8.1.2. This default (or *preconnected*) filename is of the form FORT*n*.DAT, where *n* is the I/O unit.

The DEFINE FILE statement also establishes the integer variable *asv* as the *associated variable* of a file. At the end of each direct access input/output operation, *asv* is updated to contain the record number of the record immediately following the one just read or written. This means that by using the associated variable as the record number specifier, the programmer can access the records in the file sequentially. For example:

```
DEFINE FILE 3 (1000, 48, U, NREC)
FIND (3, REC=1)
```

The DEFINE FILE statement specifies that I/O unit 3 is to be connected to a file of 1000 fixed-length records; each record is 96 bytes long. The records are numbered sequentially from 1 to 1000 and are

unformatted. The name of the file which will be accessed through unit 3 is FORT003.DAT. The associated variable is NREC. The FIND statement (see below) positions the file at record 1, and initialises NREC. After this, any statement of the following form will read the next record in the file:

```
READ (3,REC=NREC) VAR
```

D.3 Record selection

The usual method for selecting the record of a direct access file which one wishes to access is to use the REC= specifier. For example:

```
READ (UNIT=10, REC=IRECNO) IARRAY
```

In this case, the number of the record to be accessed is in the variable IRECNO. This method is described in more detail in section 16.4.1.

Parallel Fortran also supports another method for selecting records. For example:

```
READ (10'IRECNO) IARRAY
```

In this example, which has the same effect as the previous one, the record selector is placed after the unit specifier, separated by a single quote character ('). The same technique may be used with WRITE and FIND.

D.4 The FIND Statement

The FIND statement can be used to position a direct access file at a particular record, and to set the file's associated variable to that record's number. The statement has the following formats:

```
FIND (u'rn,ERR=err,IOSTAT=ios)
FIND (UNIT=u, REC=rn, ERR=err, IOSTAT=ios)
```

where:

u	is an I/O unit number, which must specify a direct access file.
rn	is a record selector, which must specify a record within the file.
err	is a label, to which control is transferred if an error happens. See section 16.3.1.1.
ios	is an integer variable where an I/O status code is placed. See section 16.3.1.1.

The **IOSTAT** and **ERR** clauses, with their preceding commas, may be omitted. No I/O list may be specified and no transfer takes place. For an example of the use of the **FIND** statement, see the discussion of **DEFINE FILE** in section D.2 above.

Appendix E

Intrinsic Functions

E.1 ANSI Standard Intrinsic Functions

E.1.1 Rounding

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Truncation (see note 1)	AINT	AINT	1	Real	Double
		DINT	1	Double	Double
Nearest Integer (see note 2)	ANINT	ANINT	1	Real	Real
		DNINT	1	Double	Double
	NINT	NINT	1	Real	Integer
		IDNINT	1	Double	Integer

E.1.2 Character Type Conversion

Definition	Name	No. of Arguments	Type of Argument	Type of Function
Integer to Character	CHAR	1	Integer	Character
		1	Byte	Character
Character to Integer	ICHAR	1	Character	Integer

E.1.3 Numeric Type Conversion

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Integer	**INT**	**INT**	1	Real	Integer
		IFIX	1	Real	Integer
		IDINT	1	Double	Integer
			1	Complex	Integer
			1	DComplex	Integer
			1	Integer	Integer
Real	**REAL**	**REAL**	1	Integer	Real
		FLOAT	1	Integer	Real
			1	Real	Real
		SNGL	1	Double	Real
			1	Complex	Real
			1	DComplex	Real
Double Precision	**DBLE**		1	Integer	Double
		DBLE	1	Real	Double
			1	Double	Double
			1	Complex	Double
		DREAL	1	DComplex	Double
Complex	**CMPLX**		1 or 2	Integer	Complex
			1 or 2	Real	Complex
			1 or 2	Double	Complex
			1 or 2	Complex	Complex
			1 or 2	DComplex	Complex
Double Complex	**DCMPLX**		1 or 2	Integer	DComplex
			1 or 2	Real	DComplex
			1 or 2	Double	DComplex
			1 or 2	Complex	DComplex
			1 or 2	DComplex	DComplex

E.1.4 Arithmetic

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Absolute Value	ABS	IABS	1	Integer	Integer
		ABS	1	Real	Real
		DABS	1	Double	Double
		CABS	1	Complex	Real
		CDABS	1	Complex	Double
Remainder (see note 3)	MOD	MOD	2	Integer	Integer
		AMOD	2	Real	Real
		DMOD	2	Double	Double
Transfer of Sign (see note 4)	SIGN	ISIGN	2	Integer	Integer
		SIGN	2	Real	Real
		DSIGN	2	Double	Double
Positive Difference (see note 5)	DIM	IDIM	2	Integer	Integer
		DIM	2	Real	Real
		DDIM	2	Double	Double
Double Length Product		DPROD	2	Real	Double
Square Root	SQRT	SQRT	1	Real	Real
		DSQRT	1	Double	Double
		CSQRT	1	Complex	Complex
		CDSQRT	1	DComplex	DComplex

E.1.5 Maximum and Minimum

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Largest Value	MAX	MAXO	≥ 2	Integer	Integer
		AMAX1	≥ 2	Real	Real
		DMAX1	≥ 2	Double	Double
		AMAXO	≥ 2	Integer	Real
		MAX1	≥ 2	Real	Integer
Smallest Value	MIN	MINO	≥ 2	Integer	Integer
		AMIN1	≥ 2	Real	Real
		DMIN1	≥ 2	Double	Double
		AMINO	≥ 2	Integer	Real
		MIN1	≥ 2	Real	Integer

E.1.6 Complex Operations

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Imaginary Part		AIMAG	1	Complex	Real
		DIMAG	1	DComplex	Double
Complex Conjugate	CONJG	CONJG	1	Complex	Complex
		DCONJG	1	DComplex	DComplex

E.1.7 Exponential and Logarithms

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Exponential (e^x)	EXP	EXP	1	Real	Real
		DEXP	1	Double	Double
		CEXP	1	Complex	Complex
		CDEXP	1	DComplex	DComplex
Natural Logarithm ($\log_e x$)	LOG	ALOG	1	Real	Real
		DLOG	1	Double	Double
		CLOG	1	Complex	Complex
		CDLOG	1	DComplex	DComplex
Common Logarithm ($\log_{10} x$)	LOG10	ALOG10	1	Real	Real
		DLOG10	1	Double	Double

E.1.8 Trigonometrical Functions

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Cosine	COS	COS	1	Real	Real
		DCOS	1	Double	Double
		CCOS	1	Complex	Complex
		CDCOS	1	DComplex	DComplex
Sine	SIN	SIN	1	Real	Real
		DSIN	1	Double	Double
		CSIN	1	Complex	Complex
		CDSIN	1	DComplex	DComplex
Tangent	TAN	TAN	1	Real	Real
		DTAN	1	Double	Double
		CTAN	1	Complex	Complex
Arccosine	ACOS	ACOS	1	Real	Real
		DACOS	1	Double	Double
Arcsine	ASIN	ASIN	1	Real	Real
		DASIN	1	Double	Double
Arctangent (arctan x)	ATAN	ATAN	1	Real	Real
		DATAN	1	Double	Double
Arctangent (arctan($\frac{x1}{x2}$))	ATAN2	ATAN2	2	Real	Real
		DATAN2	2	Double	Double

E.1.9 Trigonometrical Functions (Degree)

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Cosine	COSD	COSD	1	Real	Real
		DCOSD	1	Double	Double
Sine	SIND	SIND	1	Real	Real
		DSIND	1	Double	Double
Tangent	TAND	TAND	1	Real	Real
		DTAND	1	Double	Double
Arccosine	ACOSD	ACOSD	1	Real	Real
		DACOSD	1	Double	Double
Arcsine	ASIND	ASIND	1	Real	Real
		DASIND	1	Double	Double
Arctangent ($\arctan x$)	ATAND	ATAND	1	Real	Real
		DATAND	1	Double	Double
Arctangent ($\arctan(\frac{x1}{x2})$)	ATAN2D	ATAN2D	2	Real	Real
		DATAN2D	2	Double	Double

These functions assume that angles are expressed in degrees, rather than radians. They are extensions to the ANSI standard.

E.1.10 Hyperbolic Functions

Definition	Generic Name	Specific Name	No. of Arguments	Type of Argument	Type of Function
Hyperbolic Cosine	COSH	COSH	1	Real	Real
		DCOSH	1	Double	Double
Hyperbolic Sine	SINH	SINH	1	Real	Real
		DSINH	1	Double	Double
Hyperbolic Tangent	TANH	TANH	1	Real	Real
		DTANH	1	Double	Double

E.1.11 Character Operations

Definition	Name	No. of Arguments	Type of Argument	Type of Function
Length of Character Entity	`LEN`	1	Character	Integer
Location of Substring a2 in string a1	`INDEX`	2	Character	Integer

E.1.12 Lexical Character Comparisons

Definition	Name	No. of Arguments	Type of Argument	Type of Function
Greater than or equal	`LGE`	2	Character	Logical
Greater than	`LGT`	2	Character	Logical
Less than or equal	`LLE`	2	Character	Logical
Less than	`LLT`	2	Character	Logical

The above functions return the value `.TRUE.` if the condition is satisfied according to the ASCII collating sequence; otherwise they return `.FALSE.`. If the operands are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of that operand.

E.2 Bit-Manipulation Functions

The intrinsic functions in this section are recognised by Parallel Fortran, but are not part of the ANSI standard.

The bits of an `INTEGER` are numbered from 0 to 31, with the low-order bit of the word being numbered 0.

E.2.1 Bitwise Logical Operations

Definition	Name	No. of Arguments	Type of Argument	Type of Function
AND	**IAND** or **AND**	2	Integer	Integer
OR	**IOR** or **OR**	2	Integer	Integer
Exclusive OR	**IEOR** or **XOR**	2	Integer	Integer
Complement	**NOT**	1	Integer	Integer

E.2.2 Single-Bit Functions

Definition	Name	No. of Arguments	Type of Argument	Type of Function
Bit Set: return value of a1 with bit a2 = 1	**IBSET**	2	Integer	Integer
Bit Clear: return value of a1 with bit a2 = 0	**IBCLR**	2	Integer	Integer
Bit Test: **.TRUE.** if bit a2 of a1 = 1	**BTEST**	2	Integer	Integer

E.2.3 Shift and Extract

Definition	Name	No. of Arguments	Type of Argument	Type of Function
Shift: a1 logically shifted a2 places. If a2 < 0 shift right; else shift left	`ISHFT`	2	Integer	Integer
Circular Shift: shift low-order a3 bits in a1 circularly a2 places. If a2 < 0 shift right; else shift left	`ISHFTC`	3	Integer	Integer
Extract: extract a field from a1, a3 bits wide with a2 as low-order bit	`IBITS`	3	Integer	Integer

Notes

1. int x

2. If $x \geq 0$ then $\text{int}(x + 0.5)$
   If $x < 0$ then $\text{int}(x - 0.5)$

3. $x1 - \text{int}(x1 \div x2) \times x2$

4. If $x2 \geq 0$ then $|x1|$
   If $x2 < 0$ then $-|x1|$

5. If $x1 > x2$ then $x1 - x2$
   If $x1 \leq x2$ then 0

Appendix F

Summary of Option Switches

F.1 Compiler Switches

Further information about compiler switches can be found in section 17.2, in the subsections specified below for each switch. In the table below, the following notations are used to describe the formats of the switches.

fn	An MS-DOS filename. It may be omitted in whole or in part; the compiler's behaviour in this case is described in section 17.2.
dir	An MS-DOS filename, which will be assumed to refer to a directory.
n	An integer. By default, this is decimal; hexadecimal integers may also be input, using the notation 16_*n*.

Switches and their arguments are not case sensitive.

/B	17.2.9 Print details of **/B** switch.
/B(*n***:***n***:***n***)**	17.2.9 Change size of internal buffers.
/C	17.2.4 Check: do not generate object file.
/D	17.2.2 Compile debug comment lines starting with 'D'.
/FB*fn*	17.2.3 Put binary object output in *fn*.
/FH*fn*	17.2.3 Put hexadecimal object output in *fn*.
/FL*fn*	17.2.3 Put listing in *fn*.
/FO*fn*	17.2.3 Identical to **/FB**.
/H	17.2.10 Equivalent to **/FH** (obsolescent). A *fn* may not be specified.
/I	17.2.8 Print the compiler's identification.
/I*dir*	17.2.6 Add *dir* to the **INCLUDE** list.
/Q	17.2.8 Suppress comments and warning on standard output.
/L	17.2.10 Equivalent to **/FL** (obsolescent). A *fn* may not be specified.
/LI	17.2.7 List **INCLUDE** files.
/LX	17.2.7 Generate cross-reference listing
/PC*n*	17.2.4 Set the number of bytes required for a function or subroutine call.
/PM*n*	17.2.4 Set the number of bytes required for specifying the module number.
/R	17.2.2 Relax: permit source lines up to 132 characters in length.
/S	17.2.4 Allocate all scalar variables to static storage.
/T4	17.2.4 Generate object code for the T4 processor.
/T8	17.2.4 Generate object code for the T8 processor.
/T8A	17.2.4 Generate special object code for the Rev A T800 processor.
/U	17.2.2 Report as errors names which are not defined explicitly or with **IMPLICIT**.
/V	17.2.8 Verbose: display progress messages.
/X	17.2.6 Discard the standard **INCLUDE** list.
/Zd	17.2.5 Output source line debugging information (this is the default behaviour).
/Zi	17.2.5 Output debugging information for variables.

F.2 Linker Switches

The format of the linker's command line and full details of all the switches are discussed in chapter 19. The following is a brief summary of the switches recognised by the linker.

Each switch starts with a slash character '/' and an identifying letter; it does not matter if this letter is given in upper case or lower case. The switches can be placed anywhere in the command line but they may not occur in indirect files. No spaces are allowed between a switch's identifying letter and the rest of the switch.

`/B`*file-name* This switch specifies that the file *file-name* is to be used in preference to the default bootstrap file. There is no default extension for *file-name*.

`/C` This switch stops the linker adding the bootstrap file to the executable file.

`/G` This switch results in the linker creating a debugger information area in the executable or library file.

`/I` This switch causes the linker to display its identity and along with various statistics about the executable file such as the code and static sizes and the maximum patch size used.

`/L` This switch makes the linker generate a library file rather than an executable file.

`/O`*optimization-symbol*
This switch gives priority to the position in the executable image of the object file which defines *optimization-symbol*.

`/O@`*optimization-file*
This switch gives priority to the position in the executable image of the object files which define

the symbols whose names are contained in the file *optimization-file*. The default extension for *optimization-file* is `.opt`.

`/P` This switch has the same effect has the `/L` switch.

`/Q` This switch suppresses all warning messages (see section 19.12).

`/Q`*n* This switch suppresses output of message *n* (see appendix H).

`/S` This switch generates a map file taking its name from the first name in the list of object files.

`/S`*map-file* This switch generates a map file called *map-file*. The default extension for *map-file* is `.map`.

`/X`*entry-point*
This switch causes the linker to use the symbol *entry-point* in preference to `INMOS.ENTRY.POINT`, which is the default.

F.3 `afserver` Switches

The file server program, **`afserver`**, is used to load programs from the MS-DOS host into the B004, and to enable programs on the B004 to communicate with the MS-DOS file system and devices. The program should be called like this:

`afserver` ***command-line redirections***

where:

command-line
is a sequence of switches and program parameters. Anything which is not recognised as a switch is treated as a program parameter. Switches are interpreted by

the **afserver**, and not passed to the program. Program parameters are passed to the program, and are ignored by the **afserver**.

redirections are used to redirect standard input and output in the usual MS-DOS way. In the case of a Fortran program, standard input and output are preconnected to units 5 and 6 respectively.

For example:

```
C>afserver -:b \tf2v1\tf.b4 /t4 test >errors.lis
```

Here, '**-:b \tf2v1\tf.b4**' is an **afserver** switch, and directs it to boot a program, in this case the Fortran compiler. '**/t4**' and '**test**' are parameters for the Fortran compiler, and '**>errors.lis**' redirects the compiler's console output to the file **errors.lis**.

Only **afserver** switches which are relevant to the Parallel Fortran environment are discussed here. Further information may found in the *Standalone Compiler Implementation Manual*[13]. Note that switches may start with '**-:**', as cited here, or '**/:**'. Switches must be specified in lower case.

-:b *file-name*

Boot transputer. The **afserver** will boot the program in *file-name* into the transputer board and start it. Normally, *file-name* will be a **.b4** file output by the linker or one of the configurers. Note that the complete file name must be specified, including the extension.

If a **-:b** switch is not used, the **afserver** assumes that the transputer board has already been booted, and will try to communicate with the program there.

-:l *link-address*

Specify link address. By default, the **afserver** uses a block of I/O addresses starting at either 150_{16} or 300_{16} to communicate with the transputer board. It decides

which by looking at the host's BIOS *Machine ID*. For all hosts except the original IBM PC, 150_{16} is used. However, the IBM PC uses these I/O addresses for other purposes, and consequently when the machine ID indicates that the host is an IBM PC, the **afserver** uses 300_{16} instead. (There are special varieties of the transputer boards to cope with this.) Unfortunately, the machine ID's of certain IBM-compatible machines (such as the Amstrad PC1512) indicate that they are IBM PCs, even though they more closely resemble the PC/AT. In this case, a `-:l #150` switch may be used to force the **afserver** to use the correct link address.

A hexadecimal *link-address* is indicated by preceding it with '#'.

`-:i` Information. The **afserver** prints out its version number, etc.

`-:o` *flags* Set program flags. The *flags* are used to set modes for program execution. At present, only two values are recognised.

- `-:o 0` The default. Locate the program's stack on the transputer's on-chip RAM.
- `-:o 1` Locate the whole of the program's stack in external (off-chip) storage, and use the on-chip RAM for the start of the program code.

More information about these flags may be found in section 3.5.

Appendix G

Syntax Error Messages

This appendix lists all the syntax error messages generated at compile time.

Section 17.6.1 discusses the format of syntax error messages. As mentioned in that section, there are three classes of syntax error. In the list below, which is in error code order, the letter preceding the number indicates which class the error belongs to: Error, Warning or Comment, indicated by E, W or C respectively.

E100 `Syntax error (at or before position` *column*`)`

E101 `First statement cannot be a continuation statement`

E102 `Columns 1 - 5 of a continuation statement must be blank`

E103 `Only 19 continuation statements allowed`

E104 `Statement incomplete`

E105 `Incomplete Hollerith constant`

E106 `Invalid character (at or before position` *column*`)`

E107 Invalid (non-graphic) character

E108 Non-numeric label

E109 Brackets not matched

E110 Invalid statement label *label*

E111 Label *label* not set

E112 Evaluation of constant expression causes overflow

E113 Constant exponentiation must be to an integer power

E114 Invalid character length (must be 1 - 32767)

E115 Invalid Hollerith constant length (must be 1 - 256)

E116 Invalid constant

E117 Invalid real constant

E118 Invalid complex constant

E120 Constant is not in the permitted range

E121 Illegal common block identifier *name*

E122 *name* is already in common

E123 *name* is a parameter

E124 Invalid length specification

E125 Integer constant required

E126 *name* is not a simple integer variable

E127 Illegal use of identifier *name*

E128 Invalid subprogram identifier *name*

E129 Invalid argument

E130 Invalid expression

E131 Expression must be of type integer

E132 Invalid combination of operands

E133 Invalid expression in a logical IF statement

E134 Invalid exponent

E135 Nested statement function reference

E136 Invalid subscript in implied-DO

E137 Invalid implied-DO index

E139 Wrong number of parameters

E140 Recursive statement function definition

E141 Invalid ENTRY identifier

E142 *name* **is a subprogram identifier**

E143 Parameter length or type is incorrect

E144 Variable name expected

E145 Invalid array dimension *dimension*

E146 *name* **is not valid as an array dimension**

E147 Nested use of a DO or implied-DO index

E148 Wrongly nested DO statements

E149 FORMAT statement label is missing

E151 Missing left bracket

E152 Missing right bracket

E153 - is only valid with a P scale factor

E154 *item* is an invalid format construction

E155 Decimal field is greater than width in item *item*

E156 Width of zero is invalid in item *item*

E157 Repetition factor is invalid in item *item*

E158 Null literal is not allowed

E159 Integer value is too large in item *item*

E160 No width field allowed in item *item*

E161 Input to a literal is not allowed

E162 Minimum digits field is greater than width in item *item*

E163 No format item preceding comma

E164 Non-repeatable edit desciptor *item*

E165 Comma required before item *item*

E166 Decimal point not allowed in item *item*

C174 Constant has too great a precision and has been truncated

C175 Variable name declared but not used

C177 Inaccessible statement

C178 *name* is not a standard FORTRAN77 intrinsic function

E179 *name* has already appeared in an INTRINSIC statement

E180 *name* is a constant identifier

E181 *name* has already appeared in an EXTERNAL statement

W183 *name* contains char and non-char items - not standard FORTRAN77

E184 *name* cannot be SAVEd (at line *line*) - it is an argument identifier

E185 *name* cannot be SAVEd (at line *line*) - it is a procedure identifier

E186 *name* cannot be SAVEd (at line *line*) - it is an item in common

E187 *name* specified in SAVE is a common block

E188 *name* specified in SAVE is not a common block

E189 *name* has assumed size but is not an argument

W191 Format specifier is a non-char array - not standard FORTRAN77

E192 Expression is not logical

W193 Use of Hollerith is not standard FORTRAN77

W194 Char data in non-char item is not standard FORTRAN77

E195 Function has not been assigned a value

E196 Adjustable dimension *dimension* to array *name* is not of type integer

E197 Entry name cannot be of type character (as function is not)

E198 Entry name must be of type character (as function is)

E199 Intrinsic function *name* may not be used as an argument

W200 Nonstandard facility

W201 Identifier *name* contains >6 characters - not standard FORTRAN77

E202 RETURN not allowed in main program

E203 Illegal transfer into IF, ELSEIF or ELSE block (at line *line*)

E204 Illegal transfer into IF, ELSEIF or ELSE block (from line *line*)

E205 Illegal transfer into the range of a DO loop (at line *line*)

E206 Illegal transfer into the range of a DO loop (from line *line*)

E207 ELSE or ELSEIF statement may not follow an ELSE statement

E208 ENDIF statement missing

E209 Label on ELSE or ELSEIF statement illegally referenced at line *line*

E210 Illegal reference to label on ELSE or ELSEIF statement at line *line*

E211 *keyword* specification already given

E212 *keyword* specification not applicable

E213 Invalid unit or internal file identifier

E214 Invalid format identifier

E215 Assumed size CHARACTER operand not valid in format identifier

E216 REC= value must be an integer expression

E217 *keyword* specifier is invalid

E218 Label already referenced as an executable statement (at line *line*)

E219 END= specifier is not valid in a WRITE statement

E220 Invalid format specifier for an internal file

E221 A format specifier is required when accessing an internal file

E222 REC= specifier not valid for an internal file

E223 * is not valid as unit in *i/o* statement

E224 *keyword* value must be a statement label

E225 Label *label* refers to a non-executable statement

E226 IOSTAT= specifier invalid

E227 Label has already been set (at line *line*)

E228 Label has been set or referenced as a FORMAT label (at line *line*)

E229 ELSE or ELSEIF does not follow a block IF or ELSEIF statement

E230 ENDIF does not follow a block IF,ELSEIF or ELSE statement

E231 Upper dimension bound is less than the lower bound

E232 Subscript to array *name* is outside the declared bounds

E233 Non-numeric or zero label

E234 Invalid nesting of a DO-loop and an IF-block

E235 ENTRY statement is not allowed within a DO-loop or an IF-block

E236 Misplaced specification statement

E237 Misplaced statement function statement

E238 IMPLICIT must precede all other specification except PARAMETER

E239 *name* cannot be typed after it has appeared in a PARAMETER statement

E240 *name* is already defined as a constant identifier

E241 *name* has been defined and cannot be a constant identifier

W242 Specification of item length in bytes is not standard FORTRAN77

E243 Label parameter is not allowed in a function

E244 A subprogram is not allowed to call itself recursively

E245 *arrayname* is not an array identifier

E246 *name* is invalid in a dimension expression

E247 An array is not allowed more than 7 dimensions

E248 Array *name* with adjustable dimensions is not a parameter array

E249 Array dimensions must be of type integer

E250 *arrayname* has adjustable dimension *dimension* not in COMMON or same argument list

E251 Array *name* cannot be in COMMON and have adjustable dimensions

E252 Array *name* has an assumed size but is not a parameter

E253 Subscript must be of type integer

E254 *name* cannot be an array identifier

E255 Invalid reference to assumed size array *name*

E256 Substring position value must be of type integer

E257 Invalid substring position value

E258 Combination of DOUBLE PRECISION and COMPLEX operands not allowed

E259 Invalid concatenation (includes a scalar with (*) length)

E260 Invalid comparison between arithmetic and non-arithmetic values

E261 Complex operands not permitted for .LT.,.LE.,.GT. or .GE.

E262 *name* has already been declared as an array

E264 Wrong number of subscripts specified for *name*

E265 An EQUIVALENCE list must contain at least two items

E266 An EQUIVALENCE list can contain only one COMMON item

E267 Contradiction in EQUIVALENCE list at line *line*

E268 EQUIVALENCE attempts to extend COMMON backwards

E269 *name* **has already been typed**

E270 *name* **must not appear in a type statement**

E271 Char length specification must be an integer constant expression

E272 *name* **is not permitted to have a length of (*)**

E273 Invalid alphabetic sequence

E274 Implicit type for *name* **already specified**

E275 *name* **has not been defined as a symbolic constant identifier**

E276 *name* **cannot be defined as a symbolic constant identifier**

E277 Hollerith constant exceeds item size

E278 Constant expression is of the wrong type

E279 *name* **is not valid in an INTRINSIC statement**

E280 *name* **is not a common block identifier**

E282 COMMON may only be initialized in a BLOCKDATA subprogram

E283 Items in blank common may not be initialized

E284 *name* **is not specified in a COMMON statement**

E285 Constant not compatible with variable *name*

E286 Too many constants specified

E287 Not enough constants specified

E288 *name* **is not an integer const identifier or an implied-DO-variable**

E289 Implied-DO-list may specify only array element names

E290 Iteration count for an implied-DO-list must be positive

E291 Label *label* may not be ASSIGNed

E292 Statement invalid after a logical IF

E293 Type of DO-variable is invalid

E294 Statement not allowed to end a DO-loop

E295 Increment of DO-loop is zero

E296 Invalid type of expression for a DO parameter

E297 *keyword* invalid in input/output list

E298 Invalid input list item

E299 Implied-DO-variable must not occur in the controlled input list

E300 Intrinsic function *name* must not be used as an actual argument

E301 *name* cannot appear in a DATA or EQUIVALENCE list

E302 Label *label* has already been used as a statement label

E303 END statement is missing

E305 BLOCKDATA must not contain any executable statements

E307 Statement too complex to compile

E308 Too many names in subprogram

E309 Too many faults in program

E310 Program too large

E313 A UNIT or FILE specifier must be provided

E314 UNIT and FILE specifiers may not both be provided

E315 A UNIT specifier is required

E316 A main program unit has already been compiled

E317 Common area *name* too large

E321 Statement too complex

E323 Wrong number of arguments in call of *name* at line *line*

E324 Non-corresponding subprogram type in call of *name* at line *line*

E325 Non-correspondence of arguments in call of *name* at line *line*

W328 *name* is not a standard FORTRAN77 intrinsic function

W330 Transfer into the range of a DO loop (at line *line*) is not standard FORTRAN77

W331 Transfer into the range of a DO loop (from line *line*) is not standard FORTRAN77

W332 *name* is assumed to be an intrinsic function

E333 *name* must be explicitly typed

E334 END DO has no corresponding DO statement

E335 INCLUDE file *name* is not available

E336 More than 10 nested INCLUDE statements

W337 Non-standard form of array reference in EQUIVALENCE

W340 Transfer into the range of a DO loop (at line *line*) is not standard FORTRAN77

W341 Transfer into the range of a DO loop (from line *line*) is not standard FORTRAN77

E342 Invalid concatenation including an item with (*) length at line *line*

W348 Use of DO WHILE is not standard FORTRAN77

W349 Use of END DO is not standard FORTRAN77

W350 Use of ampersand to identify a label parameter is not standard FORTRAN77

W351 Use of .XOR. instead of .NEQV. is not standard FORTRAN77

W353 Non-standard form of constant

E356 Repeated use of *name* in a dummy argument list

W357 Equivalence of char and non-char items is not standard FORTRAN77

E360 INCLUDE file name must be enclosed in quotes

Appendix H

Linker Error Messages

The error messages output by the linker are discussed in section 19.12.

In order to give as much useful information as possible, the linker will often expand messages by including such things as symbol names and numerical values. In the description of the messages, terms in *italics* will be replaced appropriately according to the following scheme:

filename	The name of a file, such as an object file name
module	The name of an object file or a module within a library
number	An integer value
switch	A letter used to identify a command line switch
symbol	A symbol defined or referenced by an object file or module
text	Various pieces of descriptive or illustrative text
type	A code for a specific type of transputer, such as T414

WARNING (0): data symbol *symbol* referenced as a code symbol in *module*

Description The given *symbol* has been defined as a *data symbol* but the given module references it as a *code symbol*.

User Action Check that the code symbol references are specifying the correct symbol and that the symbol has been defined appropriately.

WARNING (1): using definition of *symbol* in *module1*, ignoring duplicate in *module2*

Description The given *symbol* has already been defined by *module1*. Another definition has subsequently been found in *module2*. This latter definition will be ignored.

User Action Check that the correct definition is being used. If the second definition was the one that was really wanted, change the order of the object files in the link command so that the file (or library) containing the wanted definition comes before the unwanted definition.

FATAL ERROR (3): multiple INIT tags

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (4): multiple MAININIT tags

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (5): object file ***filename*** **is corrupt; illegal patch/***number*

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (6): object file ***filename*** **is corrupt; unknown tag/***number*

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (7): incompatible processor types; ***type1*** **in** ***module1*** **and** ***type2*** **in** ***module2***

Description Code compiled for one type of transputer may not be able to execute correctly on a different type of transputer. This error indicates that object files compiled for a processor of *type1* are being linked with object files compiled for a processor of *type2*.

User Action Decide the type of the target processor and recompile those object files which had been compiled for a different processor type. Also check that the correct run-time library has been specified.

FATAL ERROR (8): reserved symbol *symbol* **defined in** *module*

Description The linker reserves certain symbols for its own use. This error indicates that the given module has attempted to define such a symbol. The reserved symbols all start with two consecutive underline characters. Users should avoid using any symbols which start with these characters.

User Action Avoid using the reserved symbol.

FATAL ERROR (9): internal error/*number*

Description This message is issued when the linker discovers that its internal tables are in an inconsistent state.

User Action Submit a fault report to your distributor including the exact text of the error message.

WARNING (10): module *module* **refers to undefined symbol** *symbol*

Description The given module contains a reference to the given symbol. By the time all of the object files and libraries given in the command line have been examined, no module has been found which contains a definition of the symbol. Although this is really a fatal error, it is treated as a warning so that further undefined symbols may be discovered and reported. A fatal error (47 or 48) will be issued on the completion of the link operation.

User Action Check that the module which defines the symbol has been included in the link command and that the name

of the symbol has been given correctly in both the module which defines it and the module which references it.

FATAL ERROR (11): multiple main static initialization modules

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (12): entry point symbol *symbol* has not been defined

Description The given symbol has been specified as the entry-point for the program being linked, either by default (in which case the symbol will be **INMOS.ENTRY.POINT**) or explicitly using the **/X** linker switch. This error indicates that no module has defined that symbol.

User Action Check that the entry-point symbol has been specified correctly in the **/X** switch or that the list of object files to link includes one which defines the main entry point (a C **main** function, a Fortran **PROGRAM**, or a Pascal **PROGRAM**).

WARNING (13): no definition found for optimization symbol *symbol*

Description The given symbol has been nominated for optimization by means of the **/O** linker switch. The warning is issued if no definition for that symbol has been found by the end of the linking operation.

User Action Check that the symbol has the correct spelling and that the module containing its definition has been included in the list of files to be linked.

WARNING (14): cannot optimize position of debug area

Description It is not possible for the linker to optimize the position of the debug area. This area is identified by means of a reserved symbol, **__debug_area**, which is defined by the linker itself. The warning is issued if the name of the debug area is nominated in a **/O** linker switch.

User Action Remove the debug area symbol from the list of optimization symbols.

FATAL ERROR (15): no MAININIT found: language run-time library missing?

Description This error indicates that the linker has been unable to find the definition of an initialization module which is assumed by the 3L compilers. The problem is usually caused by omitting the run-time library from the list of files to be linked.

User Action Include the appropriate run-time library in the list of files to be linked.

FATAL ERROR (18): code position exceeds declared size in *module*

Description This error is usually caused by an error during the compilation of the given module.

User Action Recompile the given module and attempt the link operation again. If the fault persists, please contact your distributor.

FATAL ERROR (21): not enough memory

Description This error indicates that the linker has run out of available memory.

User Action If the memory available on the transputer board is more than 2MB use the **mempatch** program on the linker to set the correct memory size for the linker. The **mempatch** program is described in the 3L Parallel language manuals.

FATAL ERROR (22): patch over valid code in module *module*

Description When compiling the instructions used to access external objects, the compiler leaves a certain amount of room for the linker to *patch* in the actual address of the object. This error indicates that the size of the image file is such that the space left in the given module is not big enough to hold the actual address.

User Action Recompile the offending module and increase the amount of space left by the compiler for patches using the appropriate compile-time switch. Refer to the compiler's documentation for details.

FATAL ERROR (23): internal limitation -- too many references (*number***)**

Description The linker cannot complete the linking operation because the files being linked have more references to external symbols than can be held in the linker's internal tables. The linker's tables have space for approximately 128,000 external references.

User Action The only appropriate action is to reduce the number of external references by concatenating some of the original source files into a single file.

FATAL ERROR (24): internal limitation -- too many common blocks (*number*)

Description The linker's internal tables used for describing definitions of and references to Fortran common blocks have been filled and so the linker cannot complete the linking operation. The linker allows a total of approximately 64,000 such references and definitions.

User Action The only appropriate action is to attempt to concatenate some of the source programs that reference the same common blocks. For example, if five source programs refer to common block **X** then concatenating those files into a single source file and recompiling will reduce the number of references to common blocks from five (one per original source file) to one (in the resulting combined file).

FATAL ERROR (25): internal limitation -- unexpected end of file/*number*

Description This error is the result of attempting to link a corrupt object file or a file which is not an object file.

User Action Check that all of the files being linked are object files.

FATAL ERROR (26): internal limitation -- vector size (*number*) exceeds limit (*number*)

Description This error indicates that a record in the object file exceeds the maximum size allowed.

User Action Check that all files being linked together are proper object files or libraries.

FATAL ERROR (27): internal limitation -- too many optimization symbols (*number*)

Description The linker can only process a limited number of optimization symbols. The linker allows for approximately 1024 symbols. This error indicates that too many optimization symbols have been specified.

User Action Remove some optimization symbols.

FATAL ERROR (30): all object files are libraries; nothing to link

Description This error indicates that all of the files given on the command line are library files.

User Action Add an object file to the list of library files, or, if you mean to generate a new library file, use the **/L** switch.

FATAL ERROR (31): unknown processor type *type* in *module*

Description The given module has indicated that the code it contains is for a transputer of the given type. This type does not correspond to a transputer known to the linker.

User Action Recompile the offending module, specifying a known transputer type.

FATAL ERROR (32): unable to write to file *filename*

Description This error indicates that the named file cannot be created successfully.

User Action Check that the file name has been specified correctly and that the device on which the file is to reside has enough free space.

FATAL ERROR (33): internal limitation -- cannot process more than *number* object files

Description The linker can only process the limited number of object files and libraries. The linker allows for approximately 16,000 files. This error indicates that too many object files and libraries have been specified.

User Action Combine some of the object files into a single library file and use that library instead of the individual files.

FATAL ERROR (34): unable to open *filename*

Description This error is issued when the linker is unable to access a file.

User Action Check that the given file exists and that its name has been specified correctly.

FATAL ERROR (35): internal limitation -- too many modules (*number*)

Description The linker can only process a limited number of modules. The linker allows for approximately 16,000 modules. The error indicates that too many modules have been specified.

User Action Combine individual modules together at the source code level.

FATAL ERROR (36): internal limitation -- too many symbols (*number*)

Description The linker can only process a limited number of symbols. The linker allows for approximately 128,000 symbols. The error indicates that too many symbols have been specified.

User Action Remove any unnecessary external symbols from the source programs.

FATAL ERROR (37): command line: *text* expected

Description This error indicates that the command line has been incorrectly formed.

User Action Correct the command line.

FATAL ERROR (38): cannot specify output file twice

Description This error indicates that two or more commas have been found on the linker command line. A comma is used to separate the name of the linker's output file from the files to be linked, and there may only be one such output file.

User Action Check the format of the linker command line. In particular, make sure that the list of object files does not include commas.

FATAL ERROR (39): option /*switch* not recognised

Description The given switch is not a linker option.

User Action Correct the specification of the option.

FATAL ERROR (40): internal limitation -- too many nested data files

Description The linker imposes a limit on the depth to which data files (indirect files) can be nested. Currently, this limit is 5.

User Action Replace the most deeply nested references to indirect files with their contents.

FATAL ERROR (47): 1 symbol undefined

Description This error is issued at the end of a linking operation in which a single "undefined symbol" warning was produced

User Action Refer to **WARNING (10)**

FATAL ERROR (48): *number* symbols undefined

Description This error is issued at the end of a linking operation in which several "undefined symbol" warnings were produced.

User Action Refer to **WARNING (10)**

Appendix I

Run-Time Error Messages

I.1 General Input/Output Errors

The format of I/O error messages is discussed in section 17.6.4.1.

The statements `OPEN`, `CLOSE` and `INQUIRE` are classified as *Auxiliary* I/O statements; `REWIND`, `ENDFILE` and `BACKSPACE` are classified as *Positional* I/O statements.

118	`File already connected`	An attempt was made to `OPEN` a file on one unit while it was still connected to another.	`OPEN`

119	ACCESS conflict	When a file is to be connected to a unit to which it is already connected, then only the **BLANK** specifier may be re-defined. An attempt has been made to redefine the **ACCESS** specifier. This message is also used if an attempt is made to use a direct-access I/O statement on a unit which is connected for sequential I/O or a sequential statement on a unit connected for direct access I/O.	OPEN, Positional, READ, WRITE
120	RECL conflict	When a file is to be connected to a unit to which it is already connected, then only the **BLANK** specifier may be re-defined. An attempt has been made to redefine the **RECL** specifier.	OPEN
121	FORM conflict	When a file is to be connected to a unit to which it is already connected, then only the **BLANK** specifier may be re-defined. An attempt has been made to redefine the **FORM** specifier.	OPEN

122	STATUS conflict	When a file is to be connected to a unit to which it is already connected, then only the **BLANK** specifier may be re-defined. An attempt has been made to redefine the **STATUS** specifier.	OPEN
123	Invalid STATUS	**STATUS=KEEP** has been specified in a **CLOSE** statement for a unit which is connected to a scratch file.	CLOSE
125	Specifier not recognised	A specifier value defined by the user has not been recognised.	OPEN
126	Specifiers inconsistent	Within an **OPEN** statement one of the following invalid combinations of specifiers was defined by the user: • **FILENAME=** was specified when **STATUS=SCRATCH**; • **RECL=** was specified when **ACCESS=SEQUENTIAL**; • **BLANK=** was specified when **FORM=UNFORMATTED**.	OPEN
127	Invalid RECL value	The value of the RECL specifier was not a positive integer	OPEN

128	**Invalid filename**	The name of the file in an **INQUIRE** by file statement is not a valid filename.	**INQUIRE**
129	**No filename specified**	In an **OPEN** statement, the **STATUS** specifier was not **SCRATCH** or **UNKNOWN** and no filename was defined.	**OPEN**
130	**Record length not specified**	The **RECL** specifier was not defined although **ACCESS=DIRECT** was specified.	**OPEN**
132	**Value separator missing**	A complex or literal constant in the input stream was not terminated by a delimiter (that is, by a space, a comma or a record boundary).	List-directed **READ**
134	**Invalid scaling**	If d represents the decimal field of a format descriptor and k represents the current scale factor, then the ANSI Standard requires that the relationship $-d < k < d + 2$ is true when an **E** or **D** format code is used with a **WRITE** statement. This requirement has been violated.	**WRITE** with format
135	**Invalid logical value**	A logical value in the input stream was syntactically incorrect	List-directed **READ**

136	`Invalid character value`	A literal constant with the value '' (that is, an empty string) was found in the input stream; this is prohibited by the ANSI standard. This message is also used if a character constant did not begin with a quote.	List-directed **READ**
137	`Value not recognised`	An item in the input stream was not recognised	List-directed **READ**
138	`Invalid repetition value`	The value of a repetition factor found in the input stream is not a positive constant.	List-directed **READ**
139	`Illegal repetiton factor`	A repetition factor in the input stream was immeditaely followed by another repetition factor.	List-directed **READ**
140	`Invalid integer`	The current input field contained a real number when an integer was expected.	**READ** with format; List-directed **READ**
141	`Invalid real`	The current input field contained a real number which was syntactically incorrect.	**READ** with format; List-directed **READ**
143	`Invalid complex constant`	The current input field contained a complex number which was syntactically incorrect.	List-directed **READ**

148	**Invalid character**	A character has been found in the current input stream which cannot syntactically be part of the entity being assembled.	**READ** with format
150	**Literal not terminated**	A literal constant in the input file was not terminated by a closing quote before the end of the file.	List-directed **READ**
151	**Unit not defined**	The statement specified a unit which had been neither preconnected nor connected by an **OPEN** statement.	Any
152	**File does not exist**	An attempt has been made to open a file which does not exist with **STATUS=OLD**.	**OPEN**
153	**Input file ended**	All the data in the associated internal or external file has been read.	**READ**
154	**Wrong length record**	The record length defined by a **FORMAT** statement, or implied by an unformatted **READ** or **WRITE**, exceeds the defined maximum for the current input or output file.	**READ**; **WRITE**
155	**Incompatible format descriptor**	A format description was found to be incompatible with the corresponding item in the I/O list.	**READ** with format; **WRITE** with format

156	**Read after Write**	An attempt has been made to read a record from a sequential file after a **WRITE** statement.	**READ**
157	**Write after Endfile**	An attempt has been made to write a record to a sequential file after an **ENDFILE** statement.	**WRITE**
158	**Record number out of range**	The record number in a direct-access I/O statement is not a positive value, or when reading, is beyond the end of the file.	Direct-access **READ**; Direct-access **WRITE**
159	**No format descriptor for data item**	No corresponding format code exists in a **FORMAT** statement for an item in the I/O list of a **READ** or **WRITE** statement.	**READ** with format; **WRITE** with format
160	**Read after Endfile**	An attempt has been made to read a record from a sequential file which is positioned at **ENDFILE**.	**READ**
162	**No write permission**	An attempt has been made to write a file which is defined for input only.	**WRITE**
164	**Invalid channel number**	The unit specified in an I/O statement is a negative value.	Any
168	**File already exists**	An attempt has been made to **OPEN** an existing file with **STATUS=NEW**.	**OPEN**

169	`Output file capacity exceeded`	An attempt has been made to write an internal or external file beyond its maximum capacity.	`READ`; `WRITE`
171	`Invalid operation on file`	An I/O request was not consistent with the file definition; for example, attempting a `BACKSPACE` on a unit that is connected to the screen.	Positional; `READ`; `WRITE`
184	`Format text too large`	An array or character variable which is longer than 2048 characters has been specified as a run-time format.	`READ` with run-time format; `WRITE` with run-time format
188	`Value out of range`	During a numeric conversion from character to binary form a value in the input record was outside the range associated with the corresponding I/O item.	`READ` with format; List-directed `READ`
190	`File not suitable`	A file which can only support sequential file operations has been opened for direct access I/O.	`OPEN`
191	`Workspace exhausted`	Workspace for internal tables has been exhausted.	`OPEN`

193	`Not connected for unformatted I/O`	An attempt has been made to access a formatted file with an unformatted I/O statement.	Unformatted **READ**; Unformatted **WRITE**
194	`Not connected for formatted I/O`	An attempt has been made to access an unformatted file with a formatted I/O statement.	Formatted **READ**; Formatted **WRITE**
195	`Backspace not allowed`	An attempt was made to **BACKSPACE** a file which contains records written by a list-directed output statement; this is prohibited by the ANSI Standard.	**BACKSPACE**
199	`Field too large`	An item in the input stream was found to be more than 1024 characters long (this does not apply to literal constants.)	List-directed **READ**

I.2 Run-Time Format Errors

Errors of this type are discussed in section 17.6.4.2.

Error Number	Message
101	`Missing left bracket`
102	`Missing right bracket`
103	`Negative sign incorrect`
104	`Invalid format`
105	`Decimal field too wide`
106	`Format width zero invalid`
107	`Repetition factor invalid`
108	`Null literal invalid`
109	`Integer field too large`
110	`No width field allowed`
111	`Literal in input format`
112	`Minimum digits too large`
114	`Non-repeatable edit descriptor`
115	`Comma required`
116	`Decimal point not allowed`

I.3 Errors Returned by `afserver`

Errors of this type are discussed in section 17.6.4.3.

Error Number	Message
2	Filename too long
3	Invalid access method
4	Invalid open mode
5	Invalid exist mode
6	Invalid record length
7	Invalid stdstream
8	Invalid stream id
9	Invalid close option
10	No seek possible
11	Invalid record number
99	Operation failed

Appendix J

Mandelbrot Program Listings

J.1 Master Task

```
C MANDELM.F77
C
C Example program: Mandelbrot set evaluation and display
C Copyright (c) 1990 3L Ltd
C
C Master task
C
C The application
C ---------------
C
C The application consists of two tasks:
C
C (1) MANDELM (this file).   This is the master task, and runs in the
C     root transputer.
C (2) MANDELW.   This is the worker task, and runs in all the other
C     transputers of the net.
C
C The flood configurer, FCONFIG, can be used to produce an executable
C file which will automatically distribute the worker tasks across an
C arbitrary network and route work packets from the master to the
C workers.
C
C It is also possible to run the application in a single transputer.
C This will work automatically if the application is configured using
```

```
C FCONFIG.  Alternatively, a static single-transputer configuration
C could be built by hand, using CONFIG.  A suitable configuration file
C may be found in MANDEL.CFG.
C
C As well as various routines from the Parallel Fortran run-time library,
C MANDELM must be linked with the CGA primitives module, CGA.BIN.
C A file MANDELM.LNK is supplied, which may be used to link MANDELM,
C like this:
C
C    LINKT @MANDELM.LNK,MANDELM.B4
C
C Note that this file assumes you are working on a network of T4's, and
C will need to be changed for a network of T8's.
C
C Functions of the tasks
C ----------------------
C
C MANDELM is told by the user which part of the Mandelbrot set to
C evaluate.   It then breaks this up into 100 packets, and sends them
C to the network of MANDELW's.   As the results from each return, they
C displayed on the PC's screen.
C
C The format of the data packet which is sent by MANDELM to MANDELW is
C defined in COMMAND.INC, and the format of the data packet sent back to
C MANDELM by MANDELW is defined in RESULTS.INC.
C
C Internals of MANDELM
C --------------------
C
C The task contains three threads.
C
C (1) The MAIN thread.
C This runs in the main program.  First, it intialises the other two
C threads and the integer ICHAN as an internal channel.  Then it goes
C into a loop, once round for each Mandelbrot display.  For each, it
C gets instructions from the user, and then sends to the SEND thread
C details of the part of the Mandelbrot set to be evaluated, using the
C ICHAN channel.  It keeps track of completed work by examining
C TALLY_DONE, which is incremented by RECEIVE every time a RESULTS
C packet is displayed; whenever it notices that TALLY_DONE has changed,
C it updates the PC's display; and when TALLY_DONE reaches 100, MAIN
C knows that the display is complete.
C
C (2) The SEND thread.
C This waits until input arrives through the ICHAN channel, telling
C it what part of the Mandelbrot set to display.   When it arrives,
C this input goes straight into the first parts of the COMMAND data
C packet, whose format is defined in COMMAND.INC.   SEND then breaks
C the job into 100 small jobs.   For each, it adds the details to the
C COMMAND packet, and uses the F77_NET_SEND function to send it off
C to the network of MANDELW's. Notice that the SEND thread does not
C specify WHICH worker task is to do any particular job;  this is
```

```
C decided by the network of router tasks.
C
C (3) The RECEIVE thread.
C This simply waits till a packet arrives from the network of MANDELW's
C and then displays it.  Each packet contains all the necessary
C information to display it, so RECEIVE does not need to keep track of
C which packet is which.  Every time it does a display, RECEIVE
C increments TALLY_DONE, so that MAIN can tell when the whole display
C is complete.
C
C
C The SEND thread runs in this subroutine.
C
      SUBROUTINE SEND (X_INCREMENT, Y_INCREMENT)
      IMPLICIT NONE
      INTEGER X_INCREMENT, Y_INCREMENT
C
      INCLUDE 'COMMAND.INC'
      INCLUDE 'CGA.INC'
      INCLUDE 'CHAN.INC'
      INCLUDE 'NET.INC'
C
C     Channel to communicate with MAIN
      COMMON /INTERNAL_CHANNEL/ ICHAN
      INTEGER ICHAN
C
      INTEGER ICHANADDR, X, Y
C
      ICHANADDR = F77_CHAN_ADDRESS (ICHAN)
C
100   CONTINUE
C
C        Wait for instructions from MAIN thread
         CALL F77_CHAN_IN_MESSAGE (12, COMMAND, ICHANADDR)
C        Send off the packets to be done.  Each includes the top-left
C        and bottom-right corners of the area to do.  This both tells
C        the worker task what value to generate and identifies the
C        RESULTS packet when it arrives in the RECEIVE thread (since
C        there's not guarantee that the results will arrive in the
C        same order that the commands were sent out.)
         DO X = 0, CGA_LORES_XMAX-1, X_INCREMENT
            C_TLX = X
            C_BRX = X + X_INCREMENT-1
            DO Y = 0, CGA_YMAX-1, Y_INCREMENT
               C_TLY = Y
               C_BRY = Y + Y_INCREMENT-1
C              Send off the next packet
               CALL F77_NET_SEND (28, COMMAND, .TRUE.)
            END DO
         END DO
C
      GOTO 100
```

```
C
      END
C
C
C The RECEIVE thread runs in this subroutine
C
      SUBROUTINE RECEIVE (TALLY_DONE)
      IMPLICIT NONE
      INTEGER TALLY_DONE
C
      INCLUDE 'CGA.INC'
      INCLUDE 'NET.INC'
      INCLUDE 'RESULTS.INC'
C
      INTEGER LEN, X, Y, I, N, COLOUR
      LOGICAL COMPLETE
C
      INTEGER THRESH1, THRESH2, THRESH3
      COMMON /THRESHOLDS/ THRESH1, THRESH2, THRESH3
C
100   CONTINUE
C
C        Thread will wait here till a packet arrives
         CALL F77_NET_RECEIVE (LEN, RESULTS, COMPLETE)
C        The RESULTS packet includes the coordinates of the top-left
C        and bottom-right corners of the data, so we know where to
C        display them.

         I = 1
         DO Y = R_TLY, R_BRY
            DO X = R_TLX, R_BRX
               N = ICHAR (R_COUNTS(I:I))
               I = I + 1
C              Received 0 means 1; received 255 means 256; etc
               N = N + 1
C              Decide on colour from thresholds and display
               IF (N .GT. THRESH3) THEN
                  COLOUR = 3
               ELSE IF (N .GT. THRESH2) THEN
                  COLOUR = 2
               ELSE IF (N .GT. THRESH1) THEN
                  COLOUR = 1
               ELSE
                  COLOUR = 0
               ENDIF
               CALL CGA_LORES_PLOT (X, Y, COLOUR)
            END DO
         END DO
C        Increment tally of packets displayed
         TALLY_DONE = TALLY_DONE + 1
C
      GOTO 100
```

```
C
      END
C
C
C The MAIN thread runs here
C
      PROGRAM MAIN
      IMPLICIT NONE
C
      INCLUDE 'THREAD.INC'
      INCLUDE 'CHAN.INC'
      INCLUDE 'CGA.INC'
C
      INTEGER WS_SIZE
      PARAMETER (WS_SIZE=2500)
C
      EXTERNAL SEND, RECEIVE
C
C     Channel for communication with SEND thread
      INTEGER ICHAN
      COMMON /INTERNAL_CHANNEL/ ICHAN
C
C     Common block for communication with RECEIVE thread
      INTEGER THRESH1, THRESH2, THRESH3
      COMMON /THRESHOLDS/ THRESH1, THRESH2, THRESH3
C
C     Definition of shape and number of packets
      INTEGER PACKETS, X_INCREMENT, Y_INCREMENT
      PARAMETER (PACKETS     = 100,
     1           X_INCREMENT = (CGA_LORES_XMAX+1)/10,
     2           Y_INCREMENT = (CGA_YMAX+1)/10        )
C
      INTEGER ICHANADDR, SEND_WS(WS_SIZE), RECEIVE_WS(WS_SIZE),
     1        TALLY_DONE, PREVIOUS_TALLY
      REAL USER_INPUT(3), RANGE, XCOORD, YCOORD, GAP
      EQUIVALENCE (USER_INPUT(1), XCOORD),
     1            (USER_INPUT(2), YCOORD),
     2            (USER_INPUT(3), GAP)
C
C     Text mode, clear screen and sign on.
      CALL VIDEO_MODE (MONO_80COL_TEXT_MODE)
      PRINT *,'Parallel Fortran v. 2.0'
      PRINT *,'Example program: Mandelbrot set evaluation and display'
      PRINT *,'Copyright (c) 1988 3L Ltd'
      PRINT *,'This program requires a Colour Graphics Adaptor (CGA)'
      PRINT *
C
C     Initialise the channel for communication with SEND
      ICHANADDR = F77_CHAN_ADDRESS (ICHAN)
      CALL F77_CHAN_INIT (ICHANADDR)
C
C     Start the other threads
```

```
      TALLY_DONE = 0
      CALL F77_THREAD_START (SEND, SEND_WS, WS_SIZE*4,
     1                       F77_THREAD_URGENT,
     2                       2, X_INCREMENT, Y_INCREMENT)
      CALL F77_THREAD_START (RECEIVE, RECEIVE_WS, WS_SIZE*4,
     1                       F77_THREAD_URGENT,
     2                       1, TALLY_DONE)
C
100   CONTINUE
C
C        This will ensure that no other threads are using the Run-
C        Time Library (in fact, in this case they won't be, but I
C        have done it here as an example...)
         CALL F77_THREAD_USE_RTL
         PRINT '(''Input X coordinate:'',$)'
         READ *,XCOORD
         PRINT '(''Input Y coordinate:'',$)'
         READ *,YCOORD
         PRINT '(''Input Y range:     '',$)'
         READ *,RANGE
         GAP = RANGE / (CGA_YMAX+1)
         YCOORD = YCOORD + RANGE
C
         PRINT '(''Threshold 1: '',$)'
         READ *,THRESH1
         PRINT '(''Threshold 2: '',$)'
         READ *,THRESH2
         PRINT '(''Threshold 3: '',$)'
         READ *,THRESH3
C
C        Finished with Run-Time Library - release it
         CALL F77_THREAD_FREE_RTL
C
C        Into graphics (CGA low-resolution) mode
         CALL VIDEO_MODE (CGA_LORES_GRAPHICS_MODE)
C
C        Reset count of finished packets to zero: RECEIVE
C        will count it back up as packets come back
         TALLY_DONE = 0
C
C        Send instructions to SEND
         CALL F77_CHAN_OUT_MESSAGE (12, USER_INPUT, ICHANADDR)
C
C        Until all the packets have been done, just keep
C        updating the display when necessary.
         PREVIOUS_TALLY = 0
         DO WHILE (TALLY_DONE .LT. PACKETS)
            DO WHILE (TALLY_DONE .EQ. PREVIOUS_TALLY)
               CALL F77_THREAD_DESCHEDULE
            END DO
            CALL CGA_UPDATE
            PREVIOUS_TALLY = TALLY_DONE
```

```
        END DO
C       Extra display update, in case...
        CALL CGA_UPDATE
C
C       Once again, wait for the Run-Time Library; then beep and
C       wait till the user hits return
        CALL F77_THREAD_USE_RTL
        PRINT '(A1)', 7
        READ (*,*)
        CALL F77_THREAD_FREE_RTL
C
C       Clear the screen and text mode again
        CALL VIDEO_MODE (MONO_80COL_TEXT_MODE)
C
      GOTO 100
C
      END
C JF Dec 90
```

J.2 Worker Task

```
C MANDELW.INC
C
C Example program: Mandelbrot set evaluation and display
C Copyright (c) 1990 3L Ltd
C
C Worker task
C
C The application
C ---------------
C
C For details of the application in general, see the top of MANDELM.F77.
C
C Internals of MANDELW
C --------------------
C
C The task waits till a packet arrives.  This has the format defined
C in COMMAND.INC, and contains details of the portion of the Mandelbrot
C to do.  The task then does the work, and stores the results in the
C format defined in RESULTS.INC, which is then sent back to MANDELM.
C
      PROGRAM MANDELW
      IMPLICIT NONE
C
      INCLUDE 'NET.INC'
      INCLUDE 'COMMAND.INC'
      INCLUDE 'RESULTS.INC'
C
```

```
      INTEGER X, Y, COUNT, N
      REAL AC, BC, SIZE, A2, B2, A, B
      LOGICAL COMPLETE
C
100   CONTINUE
C
C        Task will wait here until packet arrives
         CALL F77_NET_RECEIVE (N, COMMAND, COMPLETE)
         N = 1
C        Scan the Y range...
         DO Y = C_TLY, C_BRY
            BC = C_YCOORD - Y*C_GAP
C           ...and scan the X range, thus covering every point
            DO X = C_TLX, C_BRX
               AC = C_XCOORD + X*C_GAP
               A = AC
               B = BC
               SIZE = 0.0
               COUNT = 0
               A2 = A*A
               B2 = B*B
C              Do calculation till >2.0 away, or COUNT reaches 256
               DO WHILE ((SIZE .LT. 4.0) .AND. (COUNT .LT. 256))
                  B = 2.0*A*B + BC
                  A = A2 - B2 + AC
                  A2 = A*A
                  B2 = B*B
                  SIZE = A2 + B2
                  COUNT = COUNT+1
               END DO
C              Stored 0 means 1; stored 255 means 256
               R_COUNTS(N:N) = CHAR(COUNT-1)
               N = N+1
            END DO
         END DO
C        Send the top-left and bottom-right coordinates back in the
C        RESULTS packet too, so that the RECEIVE thread of MANDELM
C        can identify the packet.
         R_TLX = C_TLX
         R_TLY = C_TLY
         R_BRX = C_BRX
         R_BRY = C_BRY
         CALL F77_NET_SEND (1024, RESULTS, .TRUE.)
C
      GOTO 100
C
      END
```

J.3 Command Packet Include File

```
C COMMAND.INC
C
C Example program: Mandelbrot set evaluation and display
C Copyright (c) 1990 3L Ltd
C
C Definition of format of a COMMAND packet
C
      INTEGER COMMAND(7)
      INTEGER C_TLX, C_TLY, C_BRX, C_BRY
      REAL C_XCOORD, C_YCOORD, C_GAP
      EQUIVALENCE
     1   (COMMAND(1), C_XCOORD), (COMMAND(2), C_YCOORD),
     2   (COMMAND(3), C_GAP),
     3   (COMMAND(4), C_TLX), (COMMAND(5), C_TLY),
     4   (COMMAND(6), C_BRX), (COMMAND(7), C_BRY)
```

J.4 Results Packet Include File

```
C RESULTS.INC
C
C Example program: Mandelbrot set evaluation and display
C Copyright (c) 1990 3L Ltd
C
C Definition of format of a RESULTS packet
C
      INTEGER RESULTS (256)
      INTEGER R_TLX, R_TLY, R_BRX, R_BRY
      CHARACTER*1008 R_COUNTS
      EQUIVALENCE
     1   (RESULTS(1), R_TLX), (RESULTS(2), R_TLY),
     2   (RESULTS(3), R_BRX), (RESULTS(4), R_BRY),
     3   (RESULTS(5), R_COUNTS)
```

J.5 Flood Configuration File

```
Task Master File=MandelM
Task Worker File=MandelW Data=10K
```

J.6 Static Configuration File

```
Processor Host
Processor Root

Wire ? Host[0] Root[0]

Task Afserver Ins=1 Outs=1
Task Filter   Ins=2 Outs=2 Data=10K
Task MandelM  Ins=2 Outs=2 Data=500K
Task MandelW  Ins=1 Outs=1 Stack=1K Heap=10K Opt=Stack Opt=Code

Connect ? Afserver[0] Filter[0]
Connect ? Filter[0]   Afserver[0]
Connect ? Filter[1]   MandelM[1]
Connect ? MandelM[1]  Filter[1]
Connect ? MandelM[0]  MandelW[0]
Connect ? MandelW[0]  MandelM[0]

Place Afserver Host
Place Filter   Root
Place MandelM  Root
Place MandelW  Root
```

Appendix K

ASCII Code Chart

	0x0x	0x1x	0x2x	0x3x	0x4x	0x5x	0x6x	0x7x
0xx0	NUL	DLE	␣	0	@	P	`	p
0xx1	SOH	DC1	!	1	A	Q	a	q
0xx2	STX	DC2	"	2	B	R	b	r
0xx3	ETX	DC3	#	3	C	S	c	s
0xx4	EOT	DC4	$	4	D	T	d	t
0xx5	ENQ	NAK	%	5	E	U	e	u
0xx6	ACK	SYN	&	6	F	V	f	v
0xx7	BEL	ETB	'	7	G	W	g	w
0xx8	BS	CAN	(	8	H	X	h	x
0xx9	HT	EM	)	9	I	Y	i	y
0xxA	LF	SUB	*	:	J	Z	j	z
0xxB	VT	ESC	+	;	K	[	k	{
0xxC	FF	FS	,	<	L	\	l	|
0xxD	CR	GS	-	=	M	]	m	}
0xxE	SO	RS	.	>	N	^	n	~
0xxF	SI	US	/	?	O	_	o	DEL

Bibliography

[1] *American National Standard Programming Language Fortran, ANSI X3.9-1978* American National Standards Institute, Inc, New York, 1978.

[2] *Disk Operating System Version 3.10 Reference.* International Business Machines, February 1985.

[3] *Microsoft MS-DOS User's Reference.* Microsoft Corporation, 1986. Document Number 410630013-320-R03-0686.

[4] *Disk Operating System Version 3.00 Technical Reference.* International Business Machines, May 1984.

[5] A. M. Lister, *Fundamentals of Operating Systems.* Macmillan Press, 1979. ISBN 0-333-27287-0.

[6] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation.* Prentice-Hall, 1987. ISBN 0-13-637331-3.

[7] *British Standard BS6154 : 1982: Method of Defining Syntactic Metalanguage.* British Standards Institution, 1981. ISBN 0-580-12530-0.

[8] R. S. Scowen. *An Introduction and Handbook for the Standard Syntactic Metalanguage.* National Physical Laboratory Report DITC 19/83, February 1983.

[9] *ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic.* Institute of Electrical and Electronics Engineers, 1985.

[10] *IMS T414 transputer: Engineering data.* Inmos Ltd., June 1987.

[11] *IMS T800 transputer: Preliminary Data.* Inmos Ltd., April 1987.

[12] *The transputer instruction set: a compiler writers' guide.* Inmos Ltd., February 1987. Publication number 72 TRN 119 01.

[13] *Stand alone compiler implementation manual.* Version 1.1, Inmos Ltd., July 1987.

[14] *TDS Compiler implementation manual.* Version 1.0, Inmos Ltd., November 19, 1986.

[15] Roger Shepherd. *Technical Note 1: Extraordinary use of transputer links.* Inmos Ltd., November 1986.

[16] Stephen Gee. *Technical Note 11: IMS B004 IBM PC add-in board.* Inmos Ltd., February 1987.

[17] Michael Rygol and Trevor Watson. *Technical Note 18: Connecting Inmos Links.* Inmos Ltd., April 1987.

[18] *Technical Note 3: File Service Protocol Definition.* 3L Ltd., April 1988.

[19] *Tbug User Guide.* 3L Ltd., 1989. Software version 1.0.

Index

B

C

D

E

F

L

R

S

T