Cambridge Parallel Processing

DAP Series

Program Development Under UNIX

(man003.04)

A Note To The Reader: This Manual Is In The Process Of Revision

This technical manual is an unaltered reprint in reference to the CPP DAP. The DAP was originally manufactured and supported by Active Memory Technology (AMT). All software and hardware rights to the original machine have since been acquired by Cambridge Management Corporation. The DAP product line is now manufactured & supported by Cambridge Parallel Processing (CPP); a division of Cambridge Management Corporation; and referred to as the CPP DAP. For additional information on the DAP product line please contact: Cambridge Parallel Processing, 16755 Von Karman Avenue, Suite 120, Irvine, CA 92714 Tel: 714-261-8901 or Fax: 714-261-8802

CPP endeavors to ensure that the information in this document is correct but does not accept responsibility for any error or omission.

Any procedure described in this document for operating CPP equipment should be read and understood by the operator before the equipment is used. To ensure that CPP functions without risk to safety or health, such procedures should be strictly observed by the operator.

The development of CPP products and services is continuous and published information may not be up to date. Any particular issue of a product may contain part only of the facilities described in this document or may contain facilities not described here. It is important to check the current position with CPP.

Specifications and statements as to performance in this document are CPP estimates intended for general guidance. They may require adjustment in particular circumstances and are therefore not formal offers or undertakings.

Statements in this document are not part of a contract or program product license save in os far as they are incorporated into a contract or license by express reference. Issue of this document does no entitle the recipient to access to or use of the products described, and such access or use may be subject to separate contracts or licenses.

Technical publication man003.04
(CPP filenames: pubs\avent\man\pdu\ed4\...)

First Edition: June 1987
Second Edition October 1987
Third Edition 13 October 1988
Fourth Edition 1 February 1991

CPP will be pleased to receive readers' views on the contents, organization, etc. of this publication. Please make contact at either of the addresses below:

Publications Manager
Cambridge Parallel Processing Ltd.
Centennial Court, Easthampstead Road
Bracknell, Berkshire, RG12 1JA
England, United Kingdom

Tel: 0114 344 861 024
Fax: 0114 344 305 544

Publications Manager
Cambridge Parallel Processing
16755 Von Karman Avenue Suite 120
Irvine
California, 92714, USA

Tel: (714) 261-8901
Fax: (714) 261-8802

Preface

This manual describes the development of programs to be run on an AMT DAP Series massively parallel computer connected to a host computer running the UNIX operating system.

It covers the compilation, execution and debugging of DAP software for programs written in both FORTRAN-PLUS and APAL (Array Processor Assembly Language), running on either the DAP hardware or on the simulator system. You are assumed to be familiar with one or both of these languages. They are described in the AMT publications listed in **References**, which starts at the bottom of this page.

This manual is arranged as follows. Chapter 1 gives an introduction. Chapter 2 describes the FORTRAN-PLUS compilation system. Chapter 3 describes how to run DAP programs, which includes writing the host program, the interface subroutines used by the host to communicate with the DAP and the run time options available. An example of a DAP program in FORTRAN-PLUS is given, together with host programs written in 'C' and FORTRAN. Program testing and debugging techniques are covered in chapter 4. Maintenance of DAP code libraries is discussed in chapter 5. Chapter 6 describes the APAL assembly system. Chapter 77 covers some aspects of multi-programming control on DAP hardware.

If you intend to use only FORTRAN-PLUS, you should read chapters 1, 2, 3, and 4. APAL programmers should read chapters 1, 6, 3, and 4. If you are developing a large program or subroutine library, you should also read chapter 5. Chapter 7 is mostly of interest to system managers.

The appendices contain command specifications and error codes.

References:

[1]	DAP Series: Introduction to FORTRAN-PLUS enhanced	AMT	man101
[2]	DAP Series: FORTRAN-PLUS enhanced	AMT	man102
[3]	DAP Series: APAL Language	AMT	man005
[4]	DAP Series: Parallel Data Transforms	AMT	man022

[5]	DAP Series: General Support Library	AMT	man010
[6]	DAP Series: DAP System Calls	AMT	man023

WARNINGS and cautions

WARNINGS

There may be places in this manual where AMT wants to let you know of possible problems. If you see the word **WARNING** in the margin, then we are saying that if you ignore the comment in the body of the text alongside **WARNING**, then your code will almost certainly cause an error, either at compilation or run time. The words 'you have to ... ' or 'you must ...' or 'you must not ...' elsewhere in the body of the text are meant to give you a similar warning.

Cautions

If you see the word **Caution** in the margin, then we are saying that if you ignore the comment in the body of the text alongside **Caution**, then your code may cause problems, but in any case we consider you would not be following good (or at least normal) practice. The words 'you should (not) ... ' elsewhere in the body of the text are meant to give you a similar caution.

Typographical conventions

The following typographical conventions are used in this manual:

- Names of variable, commands, functions, subroutines and files mentioned in the text are shown in **bold type face** .
- Computer screen or hard copy output is shown in a box:

```
This is an example of screen output
```

- Any input that you would type is shown in **bold type face**.

  Occasionally, what you have to type in is shown boxed, as an alternative to being shown in bold typeface .
- Text that would be replaced by other text in what you type in or what the computer outputs is shown in *italics*.

  For example, you might be asked to type the command:

  save *name*

  When you come to type the command you would replace *name* with the name of the file into which you wanted to save whatever was involved.

Similarly, a host screen display might be shown as:

```
Version n.m with SCSI HCU link
MCU code size 512 Kbytes, array size 4 Mbytes
TWON>
```

whereas, in what you would actually see on your screen, *n.m* would replaced by a number combination, such as 3.1.

- If you are asked to press a particular key on the keyboard, that key will printed in capital letters and will be enclosed in angled brackets.

  For example:

  <RETURN>

  is asking you to press the Return key.

- If you are asked to press one key whilst holding down another key, both keys will be enclosed in angled brackets, with the to-be-held-down key given first and the keys joined by a '-'.

  For example:

  <CONTROL-Z>

  is asking you to hold down the Control key and press the 'Z' key.

  Similarly:

  <CONTROL-SHIFT-Q>

  is asking you to press and hold down the Control key, then press and hold down either Shift key, and then press the 'Q' key.

command syntax

- The syntax for a command specifies optional and alternative sub-items in the command as:

  [ ] You don't need to include any of the item(s) enclosed in square brackets, but if you do, then you can include only one.

  { } You must have one – and only one – of the items enclosed in braces.

  . . . You can repeat the item (and its delimiter, if appropriate) preceding an ellipsis zero or more times; that is, the item can occur one or more times.

As an example, a hypothetical command, and the way it might be specified, is given on the next page.

The command might be specified as:

$$\left\{\begin{array}{l}\textbf{ab}\\ \textbf{ac}\\ \textbf{ad}\end{array}\right\}\left[\begin{array}{c}\textit{option}\left[\textit{, option} \ldots\right]\\ \textit{filename}\end{array}\right]$$

Possible variations of the command include:

ab

ab *option*

ab *option1*, *option2*, *option3*

ab *filename*

ac *option1*, *option2*

and so on, where *option*, *option1*, *option2*, *option3* and *filename* would be defined as appropriate to the command.

Table of Contents

Chapter 1

Introduction

The DAP (standing for 'Distributed Array of Processors') is a massively parallel computer of the SIMD type, which attaches to a host computer as a peripheral processor.

The processors in the base DAP are single bit, and are arranged in a square matrix; 32 x 32 in the case of the DAP 500 range and 64 x 64 in the DAP 600. The number of processors on one side of the square gives the *edge-size* of the DAP. *ES* is used as an abbreviation for edge-size in this manual.

Some models of the DAP – those with a C in their model numbers (for example, the DAP 610C) – are also fitted with *ES* x *ES* 8-bit co-processors, one co-processor for each 1-bit processor.

Programs which use the DAP are developed on a suitable host, and then executed on the DAP. A simulation system is also available, which lets you develop and run programs in the absence of DAP hardware.

This manual describes the DAP program development and run-time software for use on host systems running under the UNIX operating system. This software will let you produce programs to run on all the DAP series machines.

A companion manual, *DAP Series: Program Development under VAX/VMS* (man004), similarly describes program development on a VAX/VMS host.

DAP and host programs

A program which runs on the DAP is called a DAP program and runs in conjunction with a program on the host. The host program is entered first and controls the loading and subsequent execution of the DAP program, and the data transfers to and from the DAP, using special interface subroutines. More than one DAP program can be resident in the DAP at any one time.

This chapter provides a general description of how to develop a program which uses a DAP. The description assumes that you have already produced and edited the source files using the various facilities available on the host computer. Each stage in the development process is described in the chapters that follow.

1.1 Source files

You produce a DAP program from source files written in the high level language FORTRAN-PLUS (see [2], *DAP Series: FORTRAN-PLUS enhanced*), or the low-level language APAL (Array Processor Assembly Language – see [3], *DAP Series: APAL Language*), or in a mixture of both.

The host program is produced from source files which you can write in any language that is supported by the host operating system and is compatible with FORTRAN. The term FORTRAN is used throughout this manual as an abbreviation for FORTRAN 77.

1.2 Producing executable programs

You compile and link the FORTRAN-PLUS and APAL source using the DAP software development system described in this manual, to produce an executable DAP program. All of the DAP program development software runs on your host processor.

You compile the subroutines of the associated host program using the compiling systems running on the host. For details of host program development, see the program development publications issued by your host's manufacturer.

1.2.1 Host program production

You produce the host program by compiling the host source code files and linking them with AMT-supplied interface routines. The routines make possible the communication between the host and DAP programs. The routines are supplied in object code format in the library **dap**, which you can access by adding **-ldap** to your compilation command in the usual way.

For example, the following command compiles a FORTRAN host program in **jimcal.f**:

```
f77  jimcal.f -ldap
```

By default, the executable code is put in file **a.out** in the current directory. You can specify an output file, **outfile** say, with the **-o** flag:

```
f77  jimcal.f -ldap -o outfile
```

The interface routines are described in section 3.2 on page 37.

1.2.2 DAP program production

You compile and link DAP programs using **dapf** (the FORTRAN-PLUS compilation system) or **dapa** (the APAL assembly system).

The following command compiles a FORTRAN-PLUS DAP program in **dapdev.df** and generates a DAP object format (DOF) file in the current directory:

```
dapf dapdev.df
```

By default the DOF file will be named **d.out**. You can specify an output file, **dapout** say, using the **-o** flag to **dapf**:

```
dapf dapdev.df -o dapout
```

dapf is described in chapter 2, which starts on page 7. **dapa** is described in chapter 6, which starts on page 111.

1.2.3 Using the co-processors

A FORTRAN-PLUS programmer who wants to make use of the speed-up available with the 8-bit co-processors needs only to set the environment variable **DAPCP8** to yes. This variable is read by the complier, which then generates code to make use of the co-processors – without any further action from the programmer.

For more details, see section 2.1 on page 7.

1.3 Other development tools

The DAP development software also includes various tools which you might find useful when you are producing or running DAP programs. The tools are:

dapdb	The DAP interactive dump analyser.
daped	The DAP object format (DOF) file editor.
daplib	The DAP library maintenance system.
dapprof	The DAP execution profile analyser.
dapopt	The DAP run-time options controller.

dapdb is described in chapter 4, which starts on page 57; **daped** briefly in section 2.5.2.2 on page 27, , and more fully in appendix A.2, which starts on page 135; **daplib** in chapter 5, which starts on page 101; **dapprof** in section 3.5, which starts on page 54; **dapopt** in section 3.4, which starts on page 42.

When you are running DAP programs, you have available a powerful interactive debugging facility called **psam** (Program State Analysis Mode). **psam** is described in section 4.3, which starts on page 60.

1.4 Run-time diagnostic facilities

The host program has access to all the usual input, output and diagnostic facilities available in the host operating system to non-DAP programs. In particular, run-time errors in the host program are treated in the way the host operating system normally treats errors. If the host program terminates for any reason when the DAP is connected to it, an automatic call to **daprel** will occur (for more details, see section 3.2.1, on page 37).

You have access to DAP program diagnostics in either of two ways:

- Through the use of **psam**, the Program State Analysis Mode sub-system. You'll find full details of **psam** in section 4.3 on page 60. **psam** can be invoked in two ways (unless the default option for the **-e** flag in **dapopt** is not in force – see section 3.4.3 on page 45.):
    - If a run-time error occurs in your DAP program, a diagnostic report is output, and then control is passed to **psam** automatically.
    - If you put **pause** statements in your FORTRAN-PLUS or APAL code, then when they are executed, control is passed to **psam** automatically.
- Through FORTRAN-PLUS or APAL **trace** statements, which display on the standard error stream (usually on the screen) during the execution of your DAP program the DAP data items specified in your **trace** statements.

  AMT does not recommend the use of **trace** statements, as they tend to slow down execution of DAP programs, but they have been kept for compatibility with earlier versions of FORTRAN-PLUS and APAL.

If a run-time error occurs in the DAP program a diagnostic report is sent to the standard error stream. This report contains the following information:

- The nature and location of the error.
- A stack trace of the currently active subroutines and functions.
- A display of the values of the local variables (if the failing procedure is a FORTRAN-PLUS subprogram and was compiled at diagnostic level 2, see section 2.4.3.2 on page 22).

1.5 Filename conventions

The DAP software development system uses certain suffixes on filenames to identify the file contents. These are:

- **.da** For APAL source code files.
- **.dc** For Consolidator Input Format (CIF) files.
- **.df** For FORTRAN-PLUS source code files.
- **.dl** For Consolidator Input Format (CIF) library files.
- **.dr** For DAP state dump files.

You should always use the correct suffix for source code files and Consolidator Input Files. You are recommended to use **.dr** for libraries and dump files.

Examples:

```
frieda.da
```

is an APAL source file in the current directory,

```
/usr/dapprog/frieda.df
```

is a FORTRAN-PLUS source file in the directory **/usr/dapprog**.

files holding object code

There is no AMT-recommended convention for filenames holding DAP or host object code. The default values are **d.out** and **a.out**, but no suffix need be used in any names you specify.

Chapter 2

FORTRAN-PLUS compilation system

Executable DAP program files are loaded from the host and run on the DAP, the host controlling the run. The process of generating DOF files from one or more FORTRAN-PLUS source files is carried out by the FORTRAN-PLUS compilation system running on the host.

2.1 Producing FORTRAN-PLUS programs for various DAP models

The FORTRAN-PLUS compilation system can produce object code for DAP 500 or 600 series machines, with or without co-processors. Two environment variables, **DAPSIZE** and **DAPCP8**, let you specify what DAP model you want to generate object code for. You set environment variables using the command **setenv**, and delete them using **unsetenv**. For example:

```
setenv FRED xyz
```

sets the value of the environment variable **FRED** to **xyz**, and:

```
unsetenv FRED
```

deletes **FRED** from the environment. **setenv** on its own:

```
setenv
```

lists all the current environment variables, and their values.

DAPSIZE lets you specify whether you want to generate code for a DAP 500 or DAP 600 machine, and takes the value 32 or **64** (the DAP edge size).

DAPCP8, if it has the value **yes**, specifies that you want code for a DAP with co-processors.

So the commands:

```
setenv  DAPSIZE 64
setenv  DAPCP8  yes
```

tell the FORTRAN-PLUS compilation system that you want to generate code for a DAP 600C series machine.

If you do not give either **DAPSIZE** or **DAPCP8** or both a value (or set them to unrecognised values) then the default action is to generate code for a DAP 500 (**DAPSIZE** not set) or for a DAP without co-processors (**DAPCP8** not set).

You cannot mix code for different DAP edge sizes in the same DAP program. However, you can mix code for DAPs with and without co-processors, but the resultant DAP program will only run on a DAP with co-processors, and the code compiled for a DAP without co-processors will not use the co-processors.

2.2 Components of the FORTRAN-PLUS compilation system

The FORTRAN-PLUS compilation process runs entirely on the host, and can be divided into the 3 phases:

- FORTRAN-PLUS preprocessor
- FORTRAN-PLUS compiler
- Consolidator (linker)

The structure of the system is shown in figure 2.1 opposite. The command **dapf** controls all 3 phases and in the simplest case a single FORTRAN-PLUS source file is preprocessed, compiled and linked to form a runnable DAP object format (DOF) file. For example the command:

```
dapf  testprog.df
```

compiles the FORTRAN-PLUS source file and generates a DOF file with default name **d.out**.

The preprocessor phase expands tab characters, attaches any included files referred to in the FORTRAN-PLUS source code to the source files, selects or ignores source lines depending on the edge-size of the target DAP the program is run on, and incorporates the code for any Parallel Data Transform (PDT) statements specified in the source. (For details of PDTs, see [4] *DAP Series: Parallel Data Transforms.*) The compilation phase generates output files in Consolidator Input Format (CIF), one CIF file for each input file, which are then passed to the consolidator and linked together to form an executable DOF file. Options in the FORTRAN-PLUS compilation system are controlled by flags to the **dapf** command as described in later sections of this chapter.

For a summary of all the **dapf** flags, see section 2.7 on page 32.

2.3 FORTRAN-PLUS preprocessor

The FORTRAN-PLUS preprocessor takes the source files you have input and produces one continuous stream of output which is passed to the compiler. It interprets *directives* in the source files, replaces tab characters with an appropriate number of spaces, and enables the compiler to report errors by filename and line number.

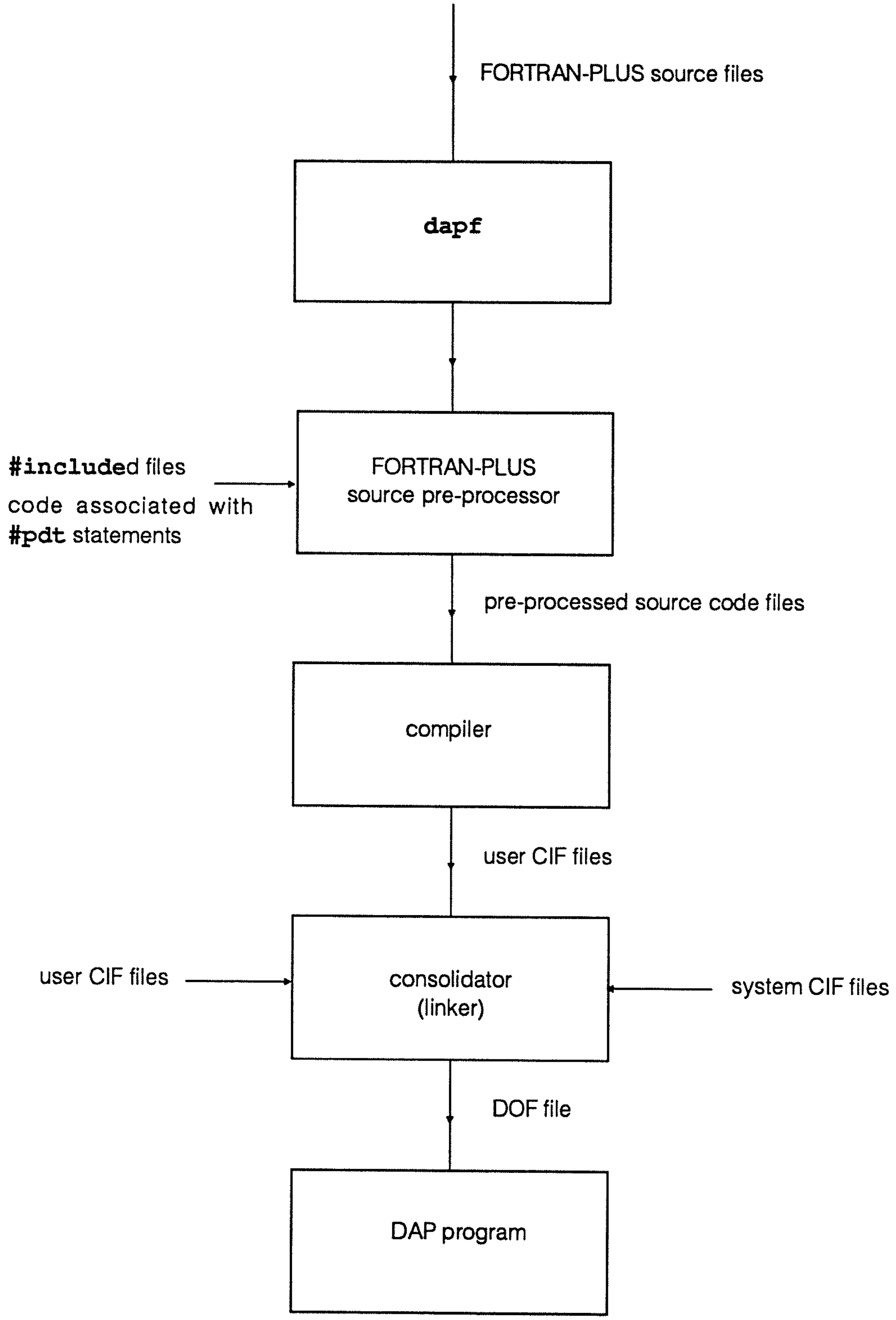

FORTRAN-PLUS source files
dapf
#included files
code associated with #pdt statements
FORTRAN-PLUS source pre-processor
pre-processed source code files
compiler
user CIF files
user CIF files
consolidator (linker)
system CIF files
DOF file
DAP program

Figure 2.1 FORTRAN-PLUS compilation system

A directive always has a # in column one, and can be one of:

- **#include**
- **#if** or **#endif**
- **#pdt**

Lines which do not start with a directive are *non-directive lines.*

2.3.1 Tab characters

FORTRAN-PLUS source code files can contain tab characters. The FORTRAN-PLUS preprocessor replaces a tab in columns 1 to 6 with one or more space characters, such that the next character in the expanded source line is in column 7. Any tabs in columns 7 onwards in the source line are each replaced with one space character.

The expanded source code should not exceed 80 characters per line. Note that characters beyond column 72 in the *expanded*line are ignored, according to the normal FORTRAN rules.

2.3.2 `#include` directive

A source code file can refer to one or more included files. An included file can also refer to one or more included files. The number of levels of nesting is restricted to 16.

A **#include** directive has three forms:

- **#include** *filename*
- **#include** "*filename*"
- **#include** <*filename*>

The first two are identical in effect. The significance of the third is explained in section 2.3.2.1 below.

A **#include** directive has to start in column 1 and cannot be continued onto the next line.

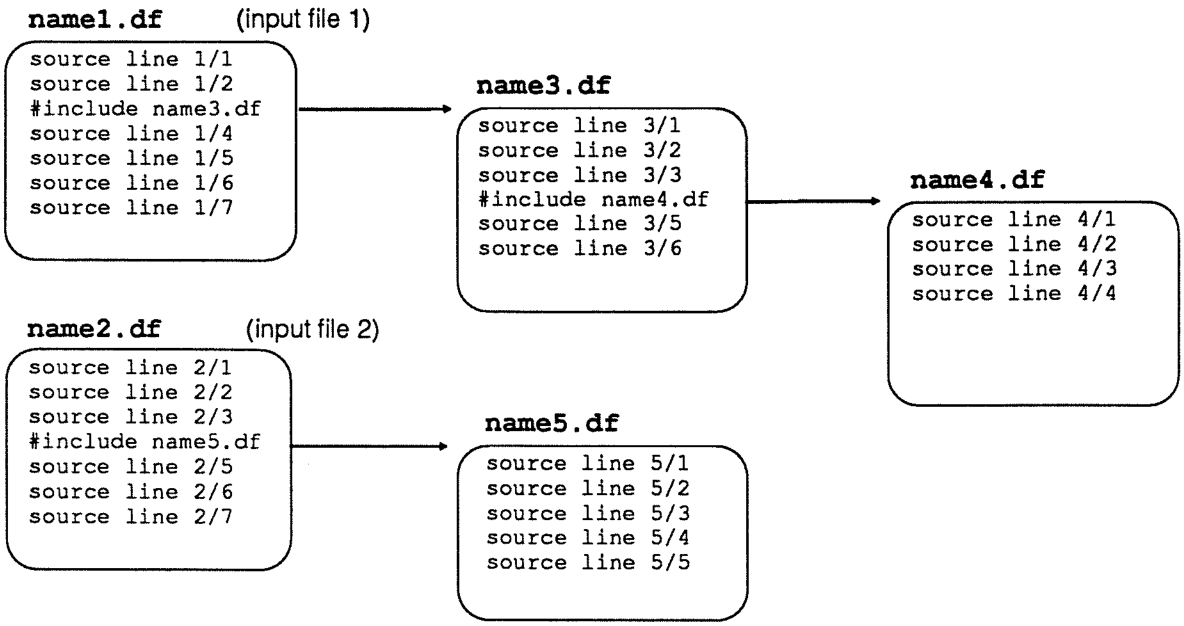
Figure 2.2 opposite shows how the **#include** directive is used, and how it affects the compiler listings. Overall line numbering might become corrupted if the file specified in a **#include** directive does not exist. For each top level source file the compiler listing line number is reset to 1.

2.3.2.1 Specifying include directories

When the preprocessor encounters a **#include** directive and the filename does not start with a **/**, a list of directories is searched in an attempt to find the file. You can add extra directories to the list by using the **-I** flag with **dapf**. This flag takes the form:

```
-I dirname
```

which adds the directory *dirname* to the list. The **-I** flag can appear more than once, so that you can add as many directories to the list as you need. The list of directories searched, and the order in which they are searched, are as follows:



name1.df (input file 1)

```
source line 1/1
source line 1/2
#include name3.df
source line 1/4
source line 1/5
source line 1/6
source line 1/7
```

name3.df

```
source line 3/1
source line 3/2
source line 3/3
#include name4.df
source line 3/5
source line 3/6
```

name4.df

```
source line 4/1
source line 4/2
source line 4/3
source line 4/4
```

name2.df (input file 2)

```
source line 2/1
source line 2/2
source line 2/3
#include name5.df
source line 2/5
source line 2/6
source line 2/7
```

name5.df

```
source line 5/1
source line 5/2
source line 5/3
source line 5/4
source line 5/5
```

The two source files are compiled and a listing is output by the compiler:

```
File:name1.df
   1:source line 1/1
   2:source line 1/2
File:name3.df  Line 1(overall line 3)
   3:source line 3/1
   4:source line 3/2
   5:source line 3/3
File:name4.df  Line 1(overall line 6)
   6:source line 4/1
   7:source line 4/2
   8:source line 4/3
   9:source line 4/4
File:name3.df  Line 5(overall line 10)
  10:source line 3/5
  11:source line 3/6
File:name1.df  Line 4(overall line 12)
  12:source line 1/4
  13:source line 1/5
  14:source line 1/6
  15:source line 1/7
File:name2.df
   1:source line 2/1
   2:source line 2/2
   3:source line 2/3
File:name5.df  Line 1(overall line 4)
   4:source line 5/1
   5:source line 5/2
   6:source line 5/3
   7:source line 5/4
   8:source line 5/5
File:name2.df  Line 5(overall line 9)
   9:source line 2/5
  10:source line 2/6
  11:source line 2/7
```

Figure 2.2 Handling multiple input source code files

1 The directory of the file containing the **#include** directive.

2 The directory of the original source file specified in the call to **dapa** or **dapf**.

3 The current working directory.

4 The list of directories specified in **dapa**'s or **dapf**'s **-I** flag, in the left-to-right order in which they appear in the call to **dapa** or **dapf**.

5 The system directories:

```
/usr/lib/dap
/usr/include
```

and then:

```
/usr/lib
```

The searching algorithm is applied in full if the **#include** directive has the form:

```
#include  frieda
```

or

```
#include  "frieda"
```

If the directive has the form:

```
#include  <frieda>
```

the search is restricted to the places detailed in 4 and 5 above.

To explain further how the preprocessor search path works, consider a simple directory hierarchy as shown in figure 2.3 opposite.

Example 1:

If the current directory is **maths**, consider the command:

```
dapf  calculation.df
```

The FORTRAN-PLUS preprocessor attempts to compile the FORTRAN-PLUS program **calculation.df** until it comes to the **#include sqroot.df** line. It then searches for the file **sqroot.df** in the current directory **maths** (items 1, 2 and 3 in the list on page 12), but fails to find it. Since, in this case the **-I** flag has not been used, the preprocessor searches the three standard directories **/usr/lib/dap**, **/usr/include** and **/usr/lib**. Unable to find **sqroot.df** in any of these directories, the preprocessor reports the failure on the standard error stream, ignores the **#include** directive and preprocesses the rest of the input file.

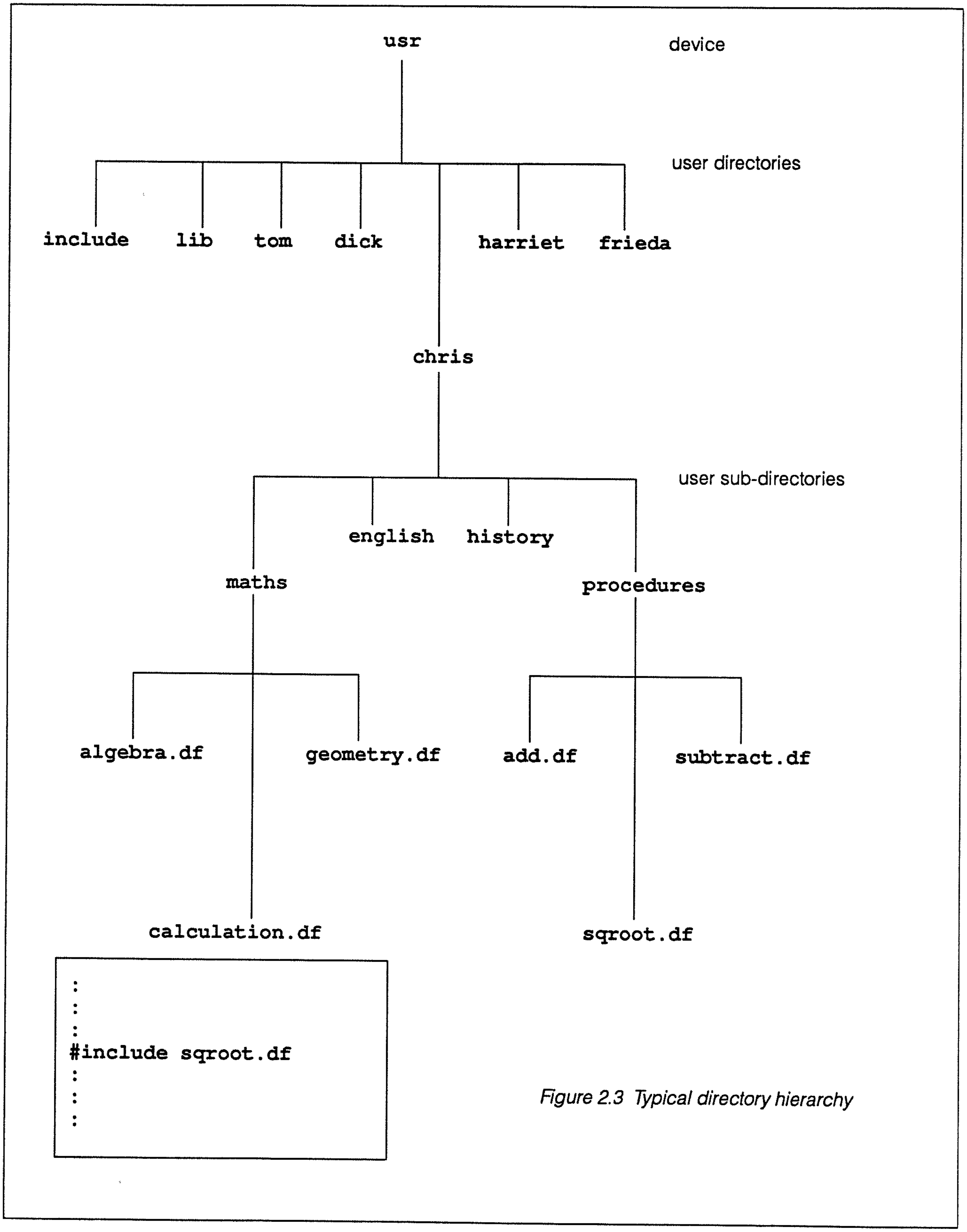


Figure 2.3 Typical directory hierarchy

Example 2:

Now consider the same command, but this time the **procedures** directory is specified using the **-I** <*dirname*> option:

```
dapf  -I ../procedures  calculation.df
```

In this case, the preprocessor will eventually find **sqroot.df** in the directory **procedures**, as it looks in the directories specified in item 4 in the list given on page 12 of the places to be searched.

2.3.3 #if and #endif directives

You can select or ignore lines from the source code files by using the **#if** and **#endif** directives.

The **#if** directive has the general form:

#if *constant expression*

There is only one form of *constant expression* currently recognised:

DAPSIZE == *value*

where:

- **DAPSIZE** is a name known to the preprocessor, and which contains either the value of the environment variable **DAPSIZE** (see section 2.1 on page 7) or the value 32
- *value* is the edge-size (*ES*) of the DAP that the program is to run on

The **#endif** directive has the form:

#endif [*comment*]

where *comment* can be omitted.

You have got to use **#if** and **#endif** directives in matching pairs; they can be nested to a depth of 16. They can appear in included files; AMT recommends that a file does not contain unmatched **#if** or **#endif** directives.

If *constant expression* is true, then the preprocessor sends non-directive lines to the compiler and processes any directive lines.

If *constant expression* is false the preprocessor ignores all subsequent lines until it finds a matching **#endif**. The exception to this rule is if the preprocessor finds **#include** directives, or further **#if** and **#endif** directives. **#include** directives will be followed to check for further **#if**s, **#endif**s or **#include**s but the non-directive lines will be ignored, until it finds a **#endif** matching the **#if** asscociated with the '*constant expression* is false'. The preprocessor will continue to ignore lines even if it comes across another **#if** directive where *constant expression* is true, until it finds a **#endif** matching the original '*constant expression* is false'.

2.3.4 #pdt directive

The **#pdt** directive lets you incorporate routines for data routing on the DAP from the Parallel Data Transform library into FORTRAN-PLUS programs.The **#pdt** directive has the form:

#pdt *pdt-statement*

pdt-statements are described in [4], *DAP Series: Parallel Data Transforms.* A **#pdt** directive can have up to 19 continuation lines, each of which has to contain a - in column 1.

The preprocessor turns a **#pdt** directive – with its continuation lines, if any – into FORTRAN-PLUS comment lines and sends them to the compiler, reformatted into 80 columns if necessary. The complete directive is then interpreted and the FORTRAN-PLUS statements are generated to invoke the Parallel Data Transform routines.

2.3.5 Preprocessor errors

The FORTRAN-PLUS preprocessor, **dapdfpp**, outputs diagnostic messages on the standard error stream. Each message is self-explanatory, and should give you a clue as to how to fix the error.

2.4 FORTRAN-PLUS compiler

2.4.1 Compiler input and output

The FORTRAN-PLUS compiler is called automatically, once the preprocessing phase has been completed. The compiler generates output in consolidator input format (CIF). By default one CIF file is created for each input source file. The CIF file has the same name as the input file, but has the file extension **.dc** instead of **.df**.

You can have all the CIF files combined into a single file by using the **-j** flag to **dapf**. For example:

```
dapf -j cif a.df b.df
```

would combine the CIF files normally placed in **a.dc** and **b.dc** into one file **cif.dc**

changing the DOF filename

The DOF file created by **dapf** has the default name **d.out**. You don't need to accept the default filename for the DOF file name, and can specify one using the **-o** flag. Hence

```
dapf -o dof a.df
```

would create the CIF file **a.dc** and link it to produce the DOF file **dof**.

You can suppress the linking phase by specifying the **-c** flag. In this case the CIF files would be produced, but no DOF file would be generated. Some time later you can link the CIF files to form a DOF file using **dapf** again, and specifying the CIF files themselves as input. Hence, if the FORTRAN-PLUS source is in several files, you only need to recompile those files which have changed.

For example, if a program consists of two FORTRAN-PLUS source files, **`a.df`** and **`b.df`**, and if **`a.dc`** has already been created, the following command would compile **`b.df`** (to generate **`b.dc`**) and then link **`a.dc`** and **`b.dc`** to produce **`d.out`**:

```
    dapf b.df a.dc
```

using `dapf` only to link CIF files

In fact, you don't need to specify *any* FORTRAN-PLUS source files when using **`dapf`**. If all input files are CIF files, then the compiler is not invoked and the consolidator is entered immediately to link all the CIF files into a single DOF program file. For example:

```
    dapf a.dc b.dc
```

will simply link the files **`a.dc`** and **`b.dc`** into the default DOF file **`d.out`**.

using `dapf` as a syntax checker

You can also suppress the linking phase by specifying the **`-y`** flag. The effect is similar to that produced by the **`-c`** flag, except that no CIF files are produced. The **`-y`** flag is useful if you want to carry out a syntax check of your FORTRAN-PLUS source without producing any CIF output.

2.4.2 Compiler listing and messages

By default the FORTRAN-PLUS compiler will not generate any compilation listings. However, you can use the **`-L`** flag to **`dapf`** to produce a brief or a full listing, according to its argument: 1 for brief, 2 for full. In both cases the listing is sent to standard output. In addition, you can use **`dapf`**'s **`-a`** flag to get a cross-reference and attribute listing, and the **`-e`** flag to get an external reference listing.

approximate position of any errors shown in listing

Diagnostic messages (comments, warnings and errors) from the compiler are also sent to standard output. The name of the file containing the offending line(s) is displayed, followed by the line(s) themselves. When an error is reported, its approximate position within the line is shown by a ^ character. Whenever diagnostic messages are generated, a one line summary of the number of comments, warnings and errors is also sent to the standard error stream. You can turn off reporting of comments by specifying the **`-q`** flag to **`dapf`**.

2.4.2.1 Brief listing

An example of a brief listing is given in figure 2.4 opposite.

A brief listing contains lines marking the start ('**`File :`** *source-filename* **`:`**' in figure 2.4) and the end (**`End of source file`** *source-filename*) of each source file. Any error messages and their associated source lines are also listed. At the end of each subprogram a summary is given, consisting of a line recording the end of the subprogram.

```
DAP FORTRAN-PLUS Compiler 4.0S (c) Copyright AMT 1987  Mon Nov  5 15:36:25 1990

Compilation for DAP 500 series
File : driver.df :

       ***COMMENT 370***
           Constant list insufficient for SUBID
           - padded with 25 * ' '
  26:       TRACE 5 (RESULTS)
                             ^
       ***COMMENT 479*** line  26 column 24
           No code generated for TRACE statement



End of compilation of subprogram - DRIVER
Subprogram diagnostics:    2 comments    0 warnings    0 errors

End of source file driver.df
Compilation summary:       2 comments    0 warnings    0 errors
```

Figure 2.4 An example of a brief listing

If compilation of the subprogram produced diagnostics, there is in addition a record of the number of comments, warnings and errors.

2.4.2.2 Full listing

The full listing consists of the pre-processed source code as it is input to the compiler, together with the information in the brief listing, as discussed above. You'll find an example of a full listing in figure 2.5 on page 18.

2.4.2.3 Cross reference and attribute listing

A cross reference and attribute listing is a list of all the names and labels used in a subprogram, except for the names of built-in functions and common block names.

The information given for each name is:

- The overall line number (that is, the line number when the source file has been expanded by any **#included** files) of the line in which the name is declared.
- The name itself.
- The class of the name. The different classes are:

Class	*Meaning*
LOCAL	A local variable
PRESET	A static local variable
PARAMETER	A parameter to the subprogram
COMMON	A variable in a common block

```
DAP FORTRAN-PLUS Compiler 4.0S (c) Copyright AMT 1987  Mon Nov  5 14:40:39 1990

Compilation for DAP 500 series

File : driver.df :
    1:       entry subroutine driver
    2:
    3:       common /data/ data
    4:       common /results/ results
    5:
    6:       integer*2 data(,)
    7:       real*4    results(,)
    8:
    9:       integer monitor
   10:       character sub_id() /'driver.'/
   11:
   12:       external real*4 matrix function clean_up
   13:
       ***COMMENT 370***
           Constant list insufficient for SUBID
           - padded with 25 * ' '
   14:       call convfm2 (data)
   15:
   16:       call process (data, results, monitor)
   17:       if (monitor.gt.0) goto 999
   18:
   19:       call convmfe (results)
   20:       return
   21:
   22: 999   continue
   23:
   24:       call report (sub_id, monitor)
   25:       results = clean_up (results)
   26:       trace 5 (results)
                               ^
       ***COMMENT 479*** line  26 column 24
           No code generated for TRACE statement

   27:       return
   28:
   29:       end

End of compilation of subprogram - DRIVER
Subprogram diagnostics:    2 comments    0 warnings    0 errors

End of source file driver.df
Compilation summary:       2 comments    0 warnings    0 errors
```

Figure 2.5 An example of a full listing

FUNCTION	A function name
SUBROUTINE	A subroutine name
BLOCK	A block data subprogram name
CONSTANT	A named constant

- The type of the item identified by the name – one of **INTEGER**, **REAL**, **LOGICAL**, or **CHARACTER** .

  Mode and rank of the item identified by the name – one of **SCALAR**, **VECTOR** or **MATRIX**; rank is given in parentheses after mode.

  For example:

  SCALAR(2) is an array of two dimensions.

  MATRIX(3) is an array of 3 dimensions, two of which are parallel.

- Length in bytes – for **INTEGER** and **REAL** items.
- A list of overall line numbers for the lines in which the name is referenced, not including the line in which the name is declared.

The information given for each label is:

- The line number in which the label is declared.
- The label itself.
- A list of line numbers in which the label is referenced, not including the line in which the label is declared.

An example of a cross reference and attribute listing is given in figure 2.6 on page 20.

2.4.2.4 External reference listing

The external reference listing is an alphabetical list of all the names in the subprogram that are defined as or assumed to be external references. As with the other listings, any diagnostic information that is output by the compiler is appended to the listing.

An example of an external reference listing is given in figure 2.7 on page 21.

2.4.2.5 Compilation diagnostics

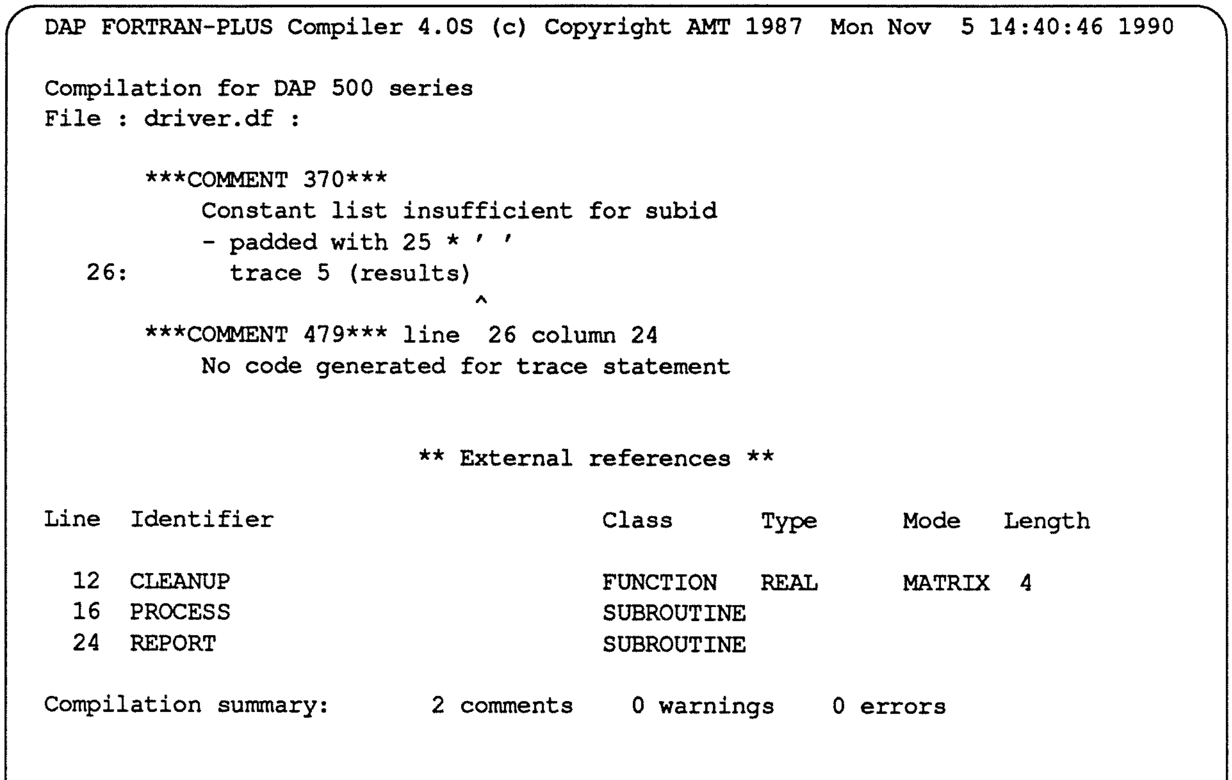
Compiler diagnostic messages are classified according to their severity level. There are four severity levels:

- **comment** Such messages indicate something that is valid in terms of the FORTRAN-PLUS language specification but is of a dubious nature. The compiler assumes that the source code is correct and continues compilation.
- **warning** Such messages indicate that something is invalid in terms of the FORTRAN-PLUS language specification but has an obvious valid interpretation. The compiler accepts this interpretation and continues compilation.

```
DAP FORTRAN-PLUS Compiler 4.0S (c) Copyright AMT 1987  Mon Nov  5 14:40:43 1990

Compilation for DAP 500 series
File : driver.df :

       ***COMMENT 370***
           Constant list insufficient for SUBID
           - padded with 25 * ' '
   26:       TRACE 5 (RESULTS)
                              ^
      ***COMMENT 479*** line  26 column 24
          No code generated for TRACE statement


                ** Cross-reference and attributes listing **

Line  Identifier                        Class      Type      Mode       Length

  12  CLEANUP                           FUNCTION   REAL      MATRIX(2)  4
            25
   3  DATA                              COMMON     INTEGER   MATRIX(2)  2
            6, 14, 16
   1  DRIVER                            SUBROUTINE

   9  MONITOR                           LOCAL      INTEGER   SCALAR     4
            16, 17, 24
  16  PROCESS                           SUBROUTINE

  24  REPORT                            SUBROUTINE

   4  RESULTS                           COMMON     REAL      MATRIX(2)  4
            7, 16, 19, 25, 25, 26
  10  SUBID                             PRESET     CHARACTER VECTOR(1)
            24

                           ** Labels listing **

Line  Label   References

  22   999    17

Compilation summary:      2 comments    0 warnings    0 errors
```

Figure 2.6 An example of a cross reference and attribute listing

- **error** Such messages indicate something invalid about which the compiler can make no valid interpretation. The compiler continues compilation, but no CIF file is produced.



```
DAP FORTRAN-PLUS Compiler 4.0S (c) Copyright AMT 1987  Mon Nov  5 14:40:46 1990

Compilation for DAP 500 series
File : driver.df :

       ***COMMENT 370***
           Constant list insufficient for subid
           - padded with 25 * ' '
   26:       trace 5 (results)
                                ^
       ***COMMENT 479*** line  26 column 24
           No code generated for trace statement


                              ** External references **

Line  Identifier                        Class      Type      Mode   Length

  12  CLEANUP                           FUNCTION   REAL      MATRIX  4
  16  PROCESS                           SUBROUTINE
  24  REPORT                            SUBROUTINE

Compilation summary:       2 comments    0 warnings    0 errors
```

Figure 2.7 An example of an external reference listing

- **`terminal`** Such messages indicate something in the system which stops the compiler from working. No CIF file is produced.

The compiler error messages should be self explanatory, and give you some idea of the source of the problem, and how to put matters right.

2.4.3 Compiler control of run-time diagnostics

Certain diagnostic features are only available at run-time if you include the appropriate code in your source code, and take the correct action at compilation time.

These features are:

- The FORTRAN-PLUS **`trace`** facility.
- The diagnostic information which is available if a run-time error occurs.
- The various run-time checks described in section 2.4.3.3 below.
- The ability to single-step through your source code.

2.4.3.1 FORTRAN-PLUS **`trace`**

All FORTRAN-PLUS **`trace`** statements have an associated level number, which can be used to control their execution at run-time (for more details see section 3.4.1, on page 44).

The level number is also used to control the conditional compilation of the **`trace`** statements. The **`-t`** flag to **`dapf`** specifies the maximum level number of **`trace`** statements which are to be compiled. The level number has to be in the range 0 to 5. At level 5 all **`trace`** statements are compiled, and at level 0 no **`trace`** statements are compiled.

Thus:

```
dapf -t2 a.df
```

will compile all level 1 and level 2 **`trace`** statements. The default value for the **`-t`** flag is zero, meaning that no **`trace`** statements are compiled.

2.4.3.2 Run-time diagnostic information

If a run-time error occurs during the execution of a DAP program, the amount of diagnostic information available will depend on the diagnostic level you specified for the compilation. The default is to give you maximum diagnostics, but you can specify a lesser level with the **`-D`** flag to **`dapf`**, which takes an argument in the range 0 – 2.

The information associated with each of the diagnostic levels is:

`0` Only subprogram names are available.

`1` As for level 0, plus line numbers.

`2` As for level 1, plus the names and values of all variables in common and static areas or currently on the stack.

The default value is 2.

2.4.3.3 Run-time checks

When a FORTRAN-PLUS program is executing, various run-time checks are normally made, as described below. By default, all the checks are carried out, but you can turn them off. You might get a slight improvement in performance as a result, but AMT does not recommend the practice unless the program has been thoroughly debugged.

You use **`dapf`**'s **`-r`** flag to switch off the check or checks, as specified by the letter(s) you give as an argument. The options are:

`c` Do not check the shape of operands for conformance.

`d` Do not check for `DO` loop increments of zero.

`n` Do not check real data for normalisation (that is, for valid internal represenation) before floating point operations.

`p` Do not check whether formal and actual parameters to routines conform in type, data-length, shape and mode.

`o` Do not check for overflow.

`s` Do not check if subscripts are in range.

`a` Do not apply any of the above-mentioned checks – that is, do not apply any run-time checks.

In addition, there is a run-time check that is not applied by default, but which you can switch on with the **+r** flag:

+rh Check whether formal and actual parameters to routines match in their non-parallel dimensions.

By default, non-parallel dimensions are not checked, but parallel dimensions are. You can turn off the checking of parallel dimensions with the **-rp** option.

Thus:

```
dapf -ra a.df
```

would compile **a.df** with no run-time checks at all, and:

```
dapf +rh -rd -rn a.df
```

would compile with the non-parallel dimensions checking switched on, and with the DO loop incrementing variable and the normalisation checks switched off.

2.4.3.4 Single-stepping

It is often of value to be able to execute only one line of source code at a time, halting program execution after each line. This *single-stepping* is available in **psam**, the program state analysis mode sub-system, provided that the source code was compiled using the **-g** flag to **dapf**.

For example, if you want to compile the FORTRAN-PLUS source code in file **a.df** and single step through the code when you enter **psam**, you could use the command:

```
dapf a.df -g
```

When you have debugged the program thoroughly, you should re-compile it without the **-g** flag.

2.4.3.5 Execution profiling

When a FORTRAN-PLUS program is executing, profiling information is generated, providing you compiled the source code using the **-p** flag to **dapf**.

This profiling information is stored in the file **dmon.out** in the current directory when the program is run. You can use the utility **dapprof** to analyse the file; **dapprof** is described in section 3.5 on page 54.

When you don't want this profiling information any more you should recompile without the **-p** flag.

2.4.3.6 Optimising

The FORTRAN-PLUS compiler lets you specify different levels of optimisation, using the **-O***n* flag to **dapf** .

Valid values for *n* are:

0 Do not carry out any optimisations.

1 Optimise the use of MCU registers and co-processor memory using simple cacheing.

2 As 1, and also carry out expression analysis, to optimise co-processor usage.

The default is **-O0**. The **-O2** option is accepted when you are compiling for non co-processor machines, although it only has an effect different from **-O1** if you are compiling for a co-processor machine.

If you specify **O** without a number, then the greatest level of optimisation available in the release of the compiler you're using will be selected.

2.5 FORTRAN-PLUS Linking

The final phase of the compilation process is to link all the consolidator input format (CIF) files (and CIF libraries, if appropriate) to form the DAP object format (DOF) file. This operation is carried out by the consolidator (also known as the linker). When **dapf** is used, the consolidator is usually entered automatically, after the compiler has finished compiling any FORTRAN-PLUS source files which you specified. The consolidator will not be invoked if you specified **dapf**'s **-c** or **-y** flags, or a compile-time error occurred.

You will get a consolidation error if any of the CIF files or libraries specified are incompatible with the current (or defaulted) values of the environment variables **DAPSIZE** and **DAPCP8** (for more details, see section 2.1 on page 7).

2.5.1 CIF library input

A CIF library is a collection of individual CIF files. You can specify one or more CIF libraries as input to **dapf**.

The consolidator will search the specified libraries for external references to subroutines and functions.

user-defined CIF libraries

CIF libraries can be user-defined or system libraries. See chapter 5, which starts on page 101, for details of how to build user-defined libraries.

User-defined libraries have the extension **.dl**. You can link in such libraries by putting the filename(s) after the input source of the CIF filename(s).

For example:

```
dapf  a.df  daplib.dl
```

will compile the FORTRAN-PLUS source code in file **a.df**, and then attempt to link it with the user-defined library **daplib.dl**, which is scanned to satisfy any external references in **a.df**.

You can link in standard system CIF libraries using the **-l** flag to **dapf**. For example, if a routine from the General Support Library (see [5], *DAP Series: General Support Library*) is referred to in **a.df**, you can access it at compile time with:

```
dapf  a.df  -l gslib
```

Coprocessor versions of system CIF libraries are used where appropriate.

The order in which the libraries and other CIF files are specified is significant, as the consolidator uses a rigid search algorithm when it is looking for unsatisfied external references – for more details, see section 5.4, starting on page 108.

2.5.2 Consolidator messages and link map

Error messages from the consolidator are sent to standard output and a summary is also sent to the standard error stream. In addition, you can ask the consolidator to display a link map on the standard output stream. This map gives information concerning the code and data areas which make up the DOF file being generated. Most users will not need to get involved with these details (at least not for simple programs). If you are a first-time reader you might want to skip this section, and carry on reading at section 2.5.3 on page 32.

The consolidator provides 3 maps: brief, standard and full. The **dapf** flag **-m** requests a map, and its argument specifies the level required (1 for brief, 2 for standard or 3 for full). By default no map is produced.

The DOF file is loaded into the memory of the DAP, when it becomes known as the user's DAP program block. There is a DAP program block for each user-program resident in the DAP.

The link map produced by the consolidator describes the memory of the DAP in terms of its code and data areas. These areas are described in the following sections, along with details of the different map options you can choose.

2.5.2.1 DAP program block

The DAP program block consists of two storage areas:

- MCU code store
- Array store

Figure 2.8 on the next page shows the format of these two storage areas.

Hardware datum and limit registers in the DAP and associated with each current user make sure that a user's DAP program cannot access areas outside its own block. The names given in figure 8 for the different areas in DAP memory are used internally by the consolidator, and some of them appear on consolidator listings.

Note that the co-processor code is stored in an area separate from the user's program block. A user doesn't normally make

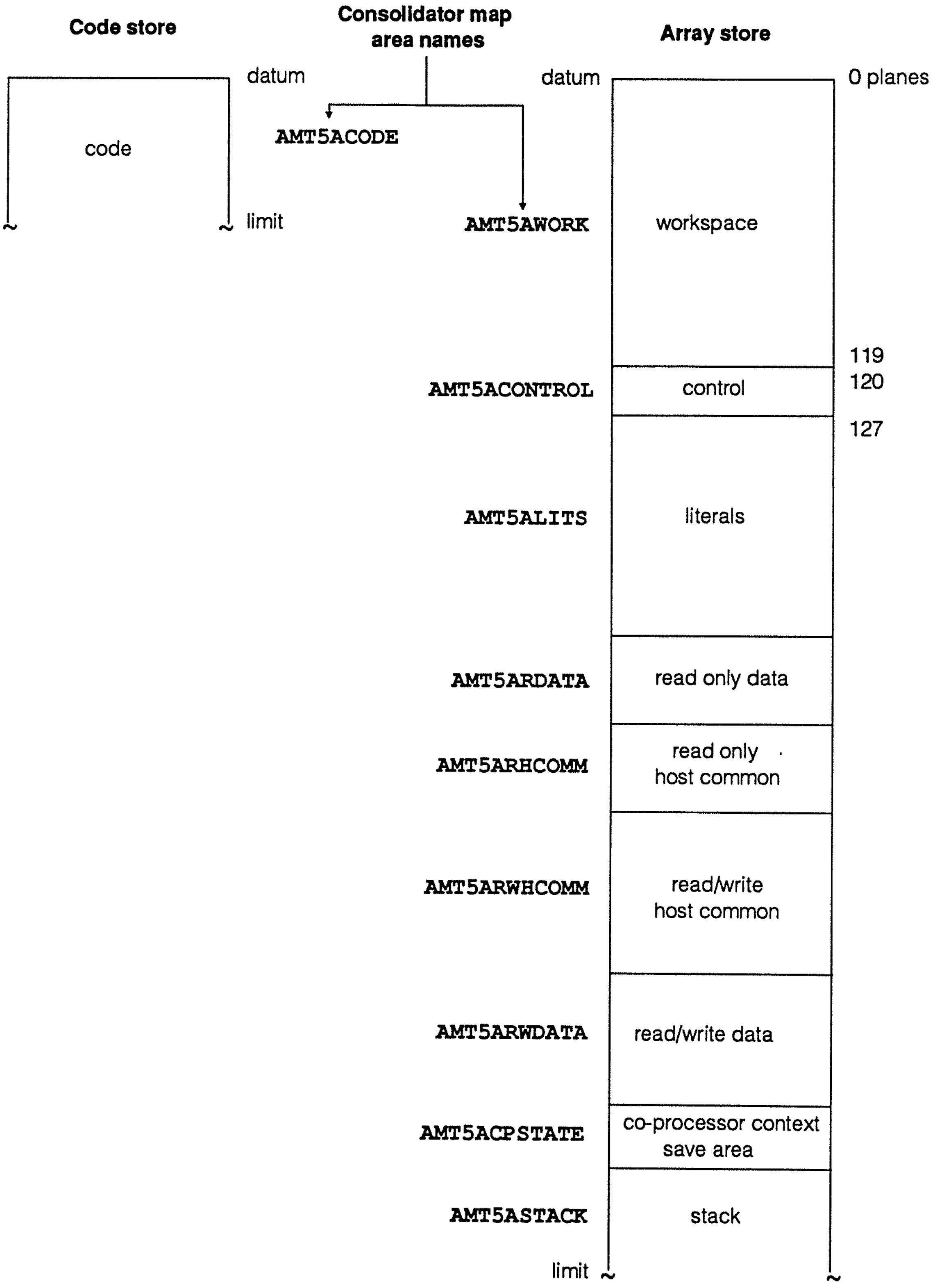


Figure 2.8 DAP program block structure

explicit use of the co-processor code store, as his program will normally only make use of system co-processor routines, which are stored in an area common to all user's programs.

The size of the stack area is set by the consolidator, which by default makes an estimate of the size needed, based on the stack requirements of the individual CIF modules which make up the DAP program.

often no need to specify required stack area

Often you don't need to specify a required stack area, since the default estimate is usually adequate, at least for FORTRAN-PLUS programs. (If you write APAL programs you are responsible for including stack request statements in each module.) However, you will sometimes find it useful to adjust the consolidator's stack estimate.

For example, if your DAP program is very large, its default stack estimate might be so large that its program allocation overflows the available space in array store, even though the program would still run correctly with a smaller stack section.

-s options to dapf

The **-s** *n* option to **dapf** instructs the consolidator to ignore its own estimate and make the stack section *n* planes in size. The **-s+***n* option to **dapf** instructs the consolidator to increase its estimate of the stack section size by *n* planes.

WARNING

Beware that if you declare your FORTRAN-PLUS variables as, say, :

```
matvar(*,*)
```

then the consolidator has no basis on which to allocate stack to your program. It makes certain assumptions about the stack requirement, but its estimate may be too low and when you come to run the program you might get a message:

```
Attempted access outside array store datum or limit
```

2.5.2.2 *Using* daped *to change stack requirements*

You can forestall this memory access problem by declaring the sizes of all your variables explicitly in your source code, or by specifying how much stack space you need.

To change the allocation of stack space to your program you could use the **-s** flag to **dapf** at link time, as discussed above.

Alternatively, once your DOF file has been produced you can use **daped** to modify the DOF file to ask for less – or more – space at run time. If, for example, you need to increase your stack allocation by 1024 planes, you could enter **daped** with the command:

```
daped DOF-file-name
```

where *DOF-file-name* is the name of your DOF file. **daped**'s command for changing stack size is **s**, so when you issue it you might get the response shown at the top of the next page.

```
daped: s
Current array size (planes)    131072
Current stack size (planes)      4096
Original stack size (planes)     4096
Enter new stack size (planes):
```

Since you want to increase your stack size by 1024 planes, from its current value of 4096 to 5120, you would enter 5120 at the prompt.

You can now exit **daped** with command **q**, and run your program again. For a formal description of the effects of all **daped**'s flags, see section A.2, on page 135.

Even if you specify your variables' sizes explicitly, you can still have problems – if the total memory size needed to hold your problem is greater than the array store size of your DAP! If this were to happen, you can either recompile, using **dapf**'s **-s** option, or use **daped** to change stack size requirement.

2.5.2.3 Brief map

A brief consolidator map gives details of:

- The areas forming the DAP program block.

  The start address and size of each area in the DAP program block is given. The names used in the map correspond to those in figure 2.8, except for the two host common areas, where the names of each constituent section are given. These sections correspond to FORTRAN-PLUS **common** blocks or APAL **data** sections with the **host common** properties.

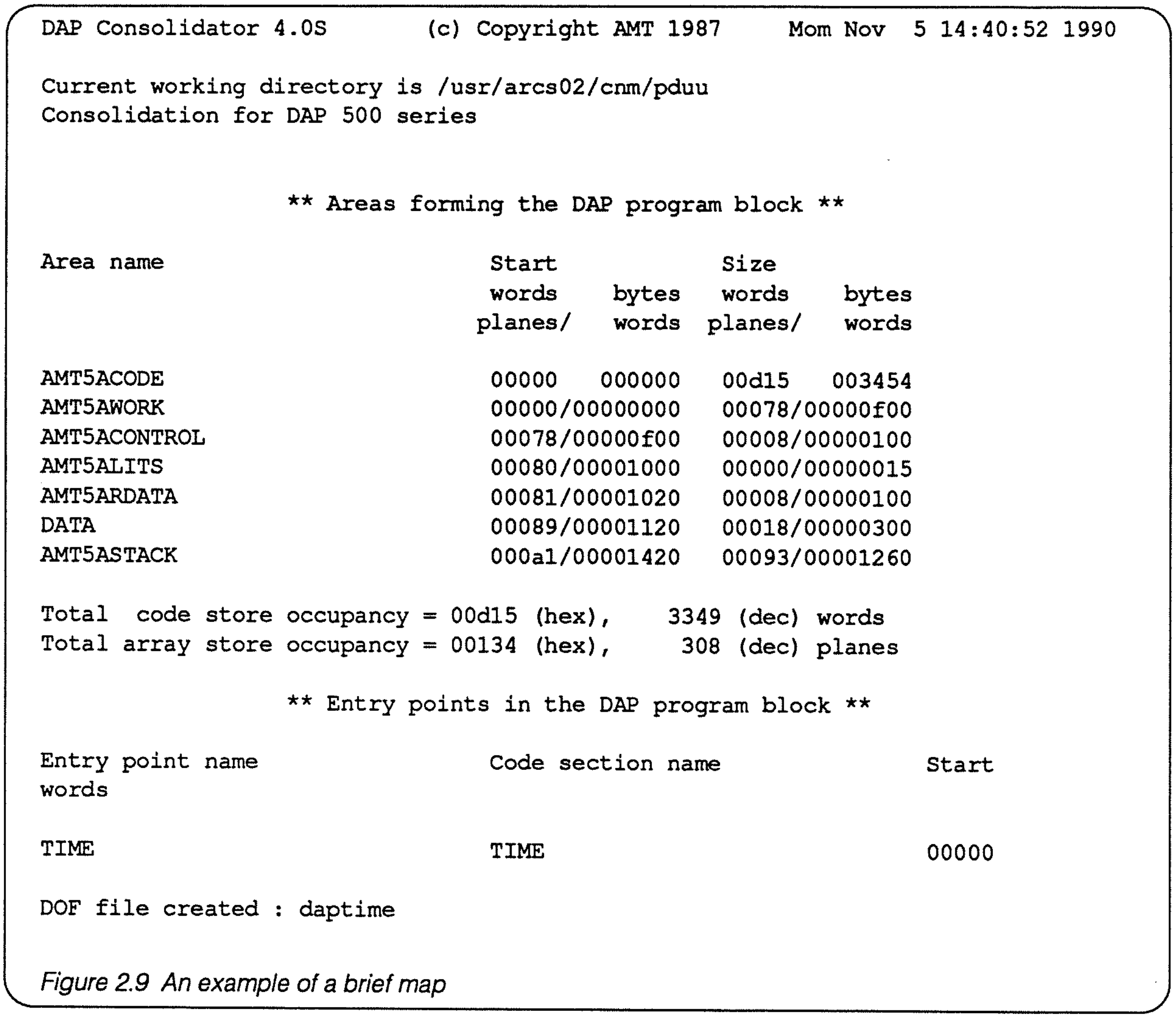
  Start points and sizes are given in bytes and words (32 bits) for MCU code store data objects and in words and planes for array store data objects, and are printed as hexadecimal values. The total array store occupied is given in planes, in hexadecimal and decimal. The total MCU code store occupied is given in words, in hexadecimal and decimal.

- Entry points in the DAP program block.

  Entry points are the places in the DAP program at which execution can start; that is, FORTRAN-PLUS **entry subroutines**, or APAL **code** sections or **entry** points with the **host** property. The start address of each entry point is given in words, in decimal, together with the name of the code section in which it occurs.

This map is sent to the standard output stream. If there are any diagnostics, a one line summary of all comments, warnings and errors is also sent to the standard error stream.

An example of a brief map is shown in figure 2.9 opposite.



```
DAP Consolidator 4.0S       (c) Copyright AMT 1987     Mom Nov  5 14:40:52 1990

Current working directory is /usr/arcs02/cnm/pduu
Consolidation for DAP 500 series


                  ** Areas forming the DAP program block **

Area name                        Start            Size
                                 words    bytes   words    bytes
                                planes/   words  planes/   words

AMT5ACODE                        00000   000000   00d15   003454
AMT5AWORK                        00000/00000000   00078/00000f00
AMT5ACONTROL                     00078/00000f00   00008/00000100
AMT5ALITS                        00080/00001000   00000/00000015
AMT5ARDATA                       00081/00001020   00008/00000100
DATA                             00089/00001120   00018/00000300
AMT5ASTACK                       000a1/00001420   00093/00001260

Total  code store occupancy = 00d15 (hex),    3349 (dec) words
Total array store occupancy = 00134 (hex),     308 (dec) planes

                  ** Entry points in the DAP program block **

Entry point name                 Code section name               Start
words

TIME                             TIME                            00000

DOF file created : daptime
```

Figure 2.9 An example of a brief map

2.5.2.4 Standard map

A standard map gives exactly the same information as the brief map, with the addition of details of the user-supplied sections in the following areas:

- MCU code area
- Read-only data area
- Read-only host common area
- Read and write host common area
- Read and write data area

An example of a standard map is shown in figure 2.10 on the next page.

In both the code and data sections of a standard map the scope of the different sections is given as **HOST**, **DAP** or **PRIV**. The meaning of these scope values is discussed below.

```
DAP Consolidator 4.0S       (c) Copyright AMT 1987     Mon Nov  5 14:40:56 1990

Current working directory is /usr/arcs02/cnm/pduu
Consolidation for DAP 500 series

                   ** Areas forming the DAP program block **

Area name                        Start            Size
                                 words    bytes   words    bytes
                                planes/   words  planes/   words

AMT5ACODE                        00000   000000   00d15   003454
AMT5AWORK                        00000/00000000   00078/00000f00
AMT5ACONTROL                     00078/00000f00   00008/00000100
AMT5ALITS                        00080/00001000   00000/00000015
AMT5ARDATA                       00081/00001020   00008/00000100
DATA                             00089/00001120   00018/00000300
AMT5ASTACK                       000a1/00001420   00093/00001260

Total  code store occupancy = 00d15 (hex),    3349 (dec) words
Total array store occupancy = 00134 (hex),     308 (dec) planes

                  ** Entry points in the DAP program block **

Entry point name                 Code section name              Start
words
TIME                             TIME                           00000

                   ** Sections in the DAP program block **

                             * Code sections *

Code section name                 Scope    Start    Size     Lits Used
 Entry point name                          words    words

TIME                              HOST     00000    0004b    0005
        module   : TIME
        CIF file : time.dc
CLOCK                              DAP     0004b    00022    0001
        module   : CLOCK
        CIF file : time.dc

                  * Read/write host common data sections *

Name                              Scope    Start     Size
                                  Common  planes    planes

DATA                              HOST Y   00089    00018
        module   : TIME
        CIF file : time.dc

DOF file created : daptime
```

Figure 2.10 An example of a standard map

The meaning of the scope values is:

- **HOST** – applies to a FORTRAN-PLUS **entry subroutine**, or an APAL **code** section or **entry** point with the **host** property.
- **DAP** – applies to a FORTRAN-PLUS **subroutine** or **function**, or an APAL **code** section or **entry** point with the **dap** property.
- **PRIV** – applies to an APAL **code** section or **entry** point with neither the **host** nor the **dap** property.

code sections

The start address and size of each code section are given in words, together with the number of literals used by the code section, and are printed as hexadecimal values. Any additional entry points to the code section are also listed with their start addresses.

The names are also given of the module containing the code section, and of the CIF file or library containing that module.

data sections

Up to four separate lists are given, corresponding to the four different data areas in array store (see figure 2.8, on page 26).

The four data areas contain:

- **Read-only data** – an APAL **data** section; either private, or with the **dap** property, or with the **dap** and **common** properties.
- **Read-only host common** – an APAL **data** section with the **host** and **common** properties.
- **Read and write host common** – a FORTRAN-PLUS **common** block, or APAL **data** section with the **host**, **common** and **write** properties.
- **Read and write data** – an APAL **data** section with the **write** property; either private, or with the **dap** property, or with the **dap** and **common** properties.

The format of each of the lists of the four data areas is the same. The start address and size of each data section is given in planes, and are printed as hexadecimal values. The scope of the name is given as **HOST**, **DAP** or **PRIV**, and a **Y** or **N** field indicates whether or not the name has the **common** property. The name of the module containing the data section and the CIF file or library containing that module is also given.

If the name of a data section has the **common** property, then that section is listed in the map only once. The consolidator lists the first occurence of the section it comes across, even though many modules may share that one **common** data section.

2.5.2.5 Full map

A full map gives exactly the same information as the standard map, with the addition of details of all AMT-supplied code sections and data sections. A list of any AMT-supplied co-processor code sections is also given. The format is the same as for a standard map.

2.5.3 Consolidator diagnostics

The consolidator generates the same classes of diagnostic messages as the compiler and the assembler, that is, comment, warning, error and terminal error messages.

2.6 Examples

example 1:

The simplest case is the compilation of a single source code file followed by linking the resulting CIF file, for example:

```
dapf  filename.df
```

The resulting CIF file is called **`filename.dc`**; the DOF file is called **`d.out`** by default. You can give the DOF file a name of your choice by using the **`-o`** flag:

```
dapf  -o myprog  filename.df
```

The DOF file produced by this command is called **`myprog`**.

example 2:

When you are compiling several source code files, you can join the resulting CIF files together and put them into one file – using the **`-j`** flag:

```
dapf  -j dapcif  dapfort1.df dapfort2.df dapfort3.df
```

This command results in one multi-module CIF file called **`dapcif.dc`**. In this case the CIF file would then be linked into a single DOF file, named **`d.out`** by default. You can inhibit the production of a DOF file by using the **`-c`** flag:

```
dapf  -j dapcif  -c dapfort1.df dapfort2.df dapfort3.df
```

example 3:

On request, the compiler and consolidator produce listings and maps. For example, if you want to get an external reference listing, a cross reference and attribute listing and a listing of the source, you could use:

```
dapf  -e  -a  -L2  dapfort.df
```

In addition, you can get a link map using the **`-m`** flag:

```
dapf  -e  -a  -L2  -m2  dapfort.df
```

2.7 `dapf` flags

This section contains a summary of all the flags available with **`dapf`**. You can put **`dapf`** flags and filenames in any order, but the consolidator searches files and CIF libraries in the sequence specified and this order may be significant (see section 5.4 on page 108).

-a Generate a cross reference and attribute listing

-c Do not link

-D*n* Generate various levels of diagnostic information that might be used in the event of run-time errors or by **dapdb**. Valid values for *n* are 0 to 2 inclusive; it controls the extent of available information:

Value of n	*Effect*
0	Subprogram names only are available.
1	As for 0, plus line numbers.
2	As for 1, plus names and values of all variables in common areas. or currently on the stack.

The default value is 2.

-e Generate an external reference listing

-g Allow single-stepping (execution of one line of source code) from within **psam** .

-I *dirname* Modify search paths for **#include** files. This option instructs the preprocessor to add *dirname* to the search path for **#include** files whose names do not begin with **/** .

-j *name* Join all CIF files into one file called *name***.dc** .

-l *name* Pass the CIF library associated with the software called *name* to the consolidator.

-L*n* Generate a source listing of the level specified by *n*. Valid values for *n* are:

Value of n	*Effect*
1	Brief listing
2	Full listing

By default, no listing is given.

-m*n* Generate a consolidator map of the level specified by *n*. Valid values for *n* are:

Value of n	*Effect*
1	Brief map
2	Standard map
3	Full map

By default, no map is given.

-o *filename* Generate an executable DAP program file called *filename* instead of the default name **d.out** .

-O*n* Carry out the optimisations specified by *n*. Valid values for *n* are:

Value of n	*Effect*
0	No optimisations.
1	MCU registers and co-processor memory optimised using simple cacheing.
2	As 1, plus expression analysis, to optimise co-processor usage.

If *n* is omitted, the highest level of optimisation available in the release of the compiler in use is selected.

By default, no optimisations are carried out.

`-p` Generate profiling information when the program is run.

`-q` Suppress compiler comment messages.

`-r`*x* Suppress run-time checks in the program specified by *x*. Valid values for *x* are:

Value of x	*Effect*
`c`	No checking for the shape of operands in expressions for conformance.
`d`	No checking whether the value of the `DO` loop increment is zero.
`n`	No checking of real data for normalisation before floating point operations are carried out.
`o`	No checking for overflow.
`p`	No checking if formal and actual parameters to routines conform in type, data-length, shape and mode.
`s`	No checking if subscripts are in range.
`a`	None of the above-mentioned checks are applied – that is, no run-time checks are applied.

By default, no checks are suppressed.

`+rh` Check if formal and actual parameters to routines match in their non-parallel dimensions.

`-s`*n* Set DOF stack record to *n* planes.

`-s+`*n* Set DOF stack record to *n* planes plus the consolidator estimate.

`-t`*n* Compile source trace statements which have a level less than or equal to *n*. Valid values for *n* are 0 to 5 inclusive. The default value is 0.

`-y` Inhibit the production of CIF files. The consolidator is not run. This option is in effect a syntax checker.

Other flags are ignored and a warning message is produced. If conflicting options are specified (such as `-L1 -L2`) the last one is used and the previous ones ignored.

Chapter 3

Running DAP programs

3.1 Introduction

Once you have compiled a DAP program, and created the resultant DOF file, you execute the program by running the associated host program on your host. The host program loads the DAP code into the DAP store and transfers data (if necessary) to the DAP before passing control to the DAP.

The DAP program will then run, with the host program suspended until the DAP program passes control back to the host in the usual way – or a run-time error occurs or a **pause** statement in the DAP program is executed. If the DAP program passes control back to the host, execution of the host program continues.

default action if run-time error or pause

If a run-time error occurs or the program is **pause**d, by default the run-time diagnostic system passes control to the program state analysis mode utility (**psam**). You can use **psam** to examine the state of the DAP when the error occurred or the program was **pause**d, and to restart the DAP program. See chapter 4 for more details of **psam** .

The default action when a run-time error occurs or a **pause** is executed is to enter **psam**, although the **-e** flag to **dapopt** lets you specify other actions. For more details of the **-e** flag see section 3.4.3 on page 45.

The method by which program control and data pass between the host and the DAP is described in section 3.2, and an example of how to compile and run a complete DAP program is given in section 3.3.

DOF file holds value of DAP edge size and co-processor requirements

The edge-size (*ES*) of the DAP machine the DAP program is to run on (the target DAP) is contained within the DOF file itself, as is the program's co-processor requirements. If you specify that the DAP program is to run on the simulator (see section 3.4, on page 42) the required simulator is invoked at run-time. However if the DAP program is to be run on the hardware, and the edge-size of the DAP which is connected to the host does not match the edge-size of the DAP in the DOF file, an error will occur. Similarly, if the DAP program needs a co-processor machine, and the DAP the program is run on is not a co-processor model, an error will occur.

DAP programs that do not use the co-processors can run on machines with or without co-processors.

A variety of options are available at run-time; these are described in section 3.4, on page 42.

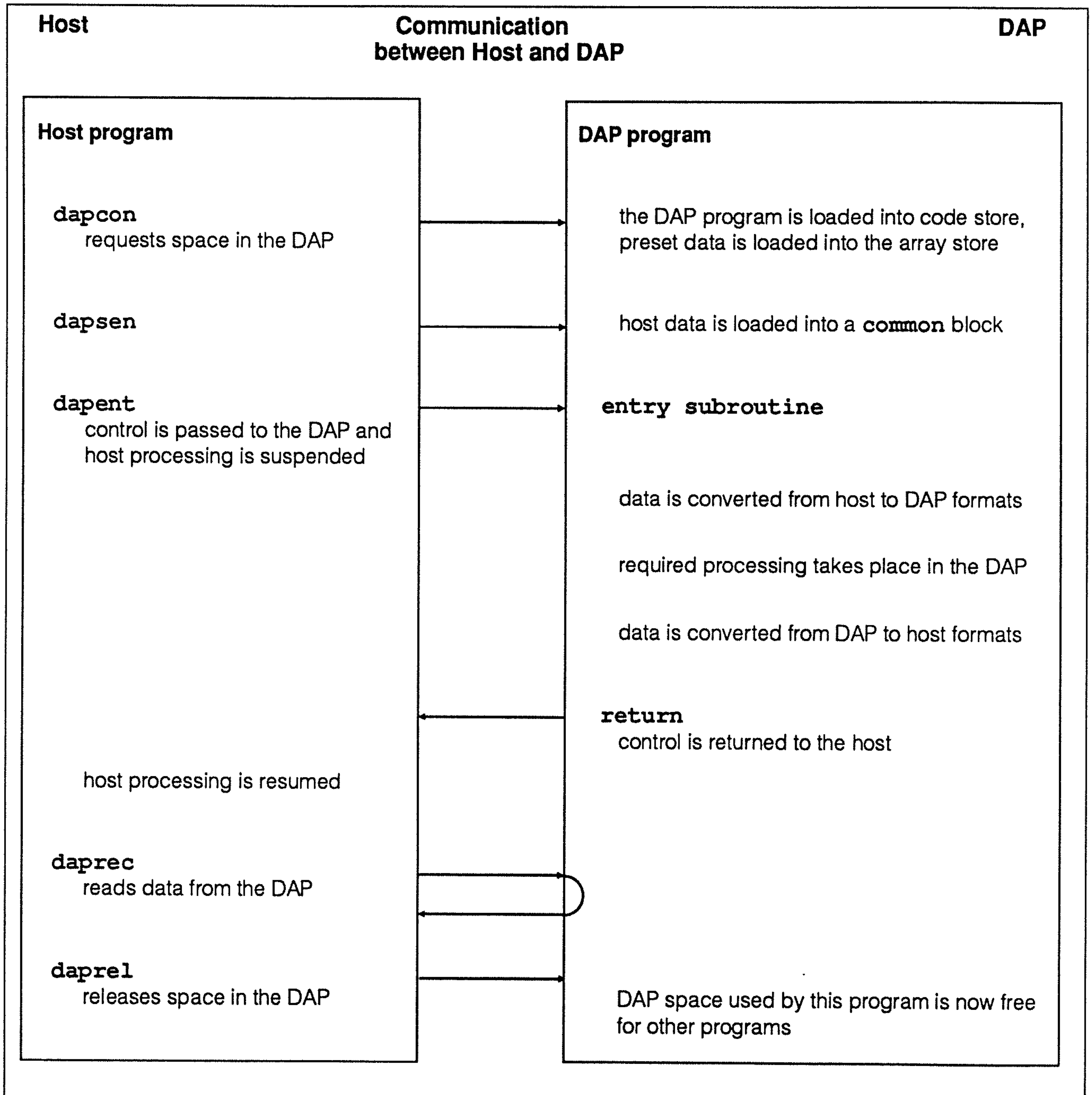

Host
Communication between Host and DAP
DAP
Host program
dapcon
requests space in the DAP
dapsen
dapent
control is passed to the DAP and host processing is suspended
host processing is resumed
daprec
reads data from the DAP
daprel
releases space in the DAP
DAP program
the DAP program is loaded into code store, preset data is loaded into the array store
host data is loaded into a common block
entry subroutine
data is converted from host to DAP formats
required processing takes place in the DAP
data is converted from DAP to host formats
return
control is returned to the host
DAP space used by this program is now free for other programs

Figure 3.1 Communication between a host program

3.2 DAP interface routines

You transfer control and data between host and DAP programs using AMT-supplied interface routines **dapcon**, **dapent**, **dapsen**, **daprec** and **daprel**. You link these routines into the host program by adding **-ldap** to your compilation command. Note that they can be called from the host program either as subroutines or functions, but the only interface routine which returns a useful result when invoked as a function is **dapcon**. In FORTRAN, you should invoke **dapcon** as a function, and declare it as **integer dapcon**. Formal specifications for the FORTRAN and C routines are given in appendix E, starting on page 161.

Figure 3.1 opposite shows the communication that takes place between the DAP program and the host program.

3.2.1 dapcon and daprel

Before a DOF file can be executed by the DAP it has to be loaded into the DAP code store. You carry out this loading process by a call to **dapcon** (DAP, connect) in your host program.

arguments to dapcon

dapcon is the first interface routine call to be issued by the host program. It is an integer function and takes one argument – the name of the file containing the DOF version of the program to be loaded. The usual form of the argument is the name of the DOF file given as a literal.

For example, in a FORTRAN host program you might issue:

result = dapcon(' *DOF-file-name* **')**

or in a C program:

result = dapcon (" *DOF-file-name* **");**

The DOF file will direct **dapcon** to the hardware or to the simulator, whichever you specified in your call to **dapopt**. If you have not called **dapopt**, the default action is to run on the hardware. For more details of **dapopt**, see section 3.4, on page 42.

running on the simulator

If in **dapopt** you specified your DAP program was to be run on the simulator, **dapcon** will load your program immediately.

running on DAP hardware

If you specified DAP hardware, then loading might not take place immediately. Several programs can be loaded into the DAP at the same time; if there is not enough space to load a new program into the DAP, the following message appears on the host screen:

```
Awaiting DAP resources
```

the program will be put in a queue, and **dapcon** will wait until there is a large enough slot in the DAP to load the program.

For more details of multi-programming on the DAP, see chapter 7, which starts on page 127.

Once your program is loaded into the DAP, **dapcon** returns control to your host program, which continues executing. **dapcon** returns an integer value in the range 0 to 5:

Result returned	*Meaning of result from* **dapcon**
0	Success
1	Unable to open DOF file
2	Unable to read DOF file
3	Not a DOF file
4	Unable to open channel to hardware DAP
5	DAP load failed

daprel

You should call **daprel** (DAP, release) from the host program when all DAP processing and communication has been completed. **daprel** should therefore always be the last DAP interface subroutine that you call from the host; it frees the DAP so that other users can access it.

arguments to daprel

daprel takes no arguments.

daprel is called automatically if the host program terminates or if you issue a second call to **dapcon** in your host program without an intervening call to **daprel**. (In this case **daprel** is first called automatically, then your second **dapcon** is acted on). If you issue a call to **daprel** when the DAP is not connected, the call is ignored.

3.2.2 dapsen and daprec

The interface subroutines **dapsen** (DAP, send to) and **daprec** (DAP, receive from) are used for the transfer of data between the host program and the DAP program.

dapsen sends data from the host program to the DAP program and **daprec** receives data from the DAP program into the host program. The use of the arguments for these two subroutines is identical, only the direction of transfer changes.

arguments to dapsen and daprec

Each subroutine takes three arguments:

name of DAP common block

- The name of a FORTRAN-PLUS **common** block (or of an APAL **data** section with the **host**, **common** and **write** properties) to or from which the data is to be sent.

start location of data to be transferred

- The name of a *word-aligned* data area in the host program (in FORTRAN programs), or the address of such an area (in C programs), identifying the start location of the block of data to be transferred.

In a FORTRAN host program

In a FORTRAN host program the normal choice for this argument is the name of the first item in a **common** block, because it will always be word-aligned. Character variables and arrays require special treatment when being transferred. Before you give the name of a character variable or array as the second argument to **dapsen** or **daprec** you should **equivalence** the character variable with an integer variable, which will make sure that the object is word-aligned. For a formal specification of the FORTRAN language interface see section E.2 on page 162.

In a C host program

In a C host program this argument should be the address of a word-aligned variable or array. If the address is not word-aligned it should be unioned to a word-aligned variable or placed in a word-aligned structure. AMT recommends that all character and short variables and arrays to be transferred to and from the DAP are actually or effectively word-aligned in this way. For a formal specification of the C language interface see section E.1 on page 161.

number of DAP words to be transferred

- An integer constant (or the name of an integer variable) which specifies the size of the data block to be transferred, in units of DAP words (a DAP word contains 32 bits)

The examples of host programs in figures 3.3 and 3.4 illustrate the use of **daprec** and **dapsen**.

Note: In your DAP program, before you use data transferred from the host, you have to call the FORTRAN-PLUS mode conversion routine **convhtod**. Similarly, if you are going to transfer data back to the host, before you leave the DAP program, you have to call **convdtoh**. For more details, see [2], *DAP Series: FORTRAN-PLUS enhanced.*

You can make any number of **dapsen** and **daprec** calls between a **dapcon** and a **daprel** call in a host program.

3.2.3 dapent

You transfer control from the host program to the DAP by a call to **dapent** (DAP, enter).

arguments to dapent

dapent takes one argument – the name of the FORTRAN-PLUS **entry subroutine** (or APAL **code** section or **entry** point with the **host** property) which is to be executed. Once the DAP program is running, execution of the host program is suspended until a **return** instruction in the DAP **entry subroutine** is obeyed. At this point the host program will start again, at the instruction immediately after the call to **dapent**.

3.3 Example

The following is a simple example of a complete FORTRAN-PLUS program, with examples of suitable host programs, one written in FORTRAN, and one in C. The example shows how to compile and run the program on the DAP hardware and on the simulator.

3.3.1 The DAP program

This example can be run on a DAP 500 or DAP 600 machine. You should set the environment variable **DAPSIZE** to the correct value for your target DAP before running **dapf** (for more details of **DAPSIZE** see section 2.1 on page 7).

Each of the two host programs initialises an array of numbers, transfers them to the DAP, which sums all the components and returns the total to the host. The example is intended to illustrate compiling and running DAP programs and host-DAP communications: it is not intended to represent an efficient use of the DAP.

The DAP program is shown in figure 3.2 below.

```
      entry subroutine dapentry
      parameter (NROWS = 40, NCOLS = 70)
      integer*4 in(*NROWS,*NCOLS)
      integer*4 isum
      common /indata/ in
      common /outdata/ isum
C
C Convert input data from host mode to matrix mode
C
      call convhtod(in)
C
C Calculate sum of matrix components
C
      isum = sum(in)
C
C Convert result from scalar mode to host mode
C
      call convdtoh(isum)
C
C Return control to the host program
C
      return
      end
```

Figure 3.2 Example DAP program

The FORTRAN-PLUS code is compiled using the **dapf** command:

```
dapf  -o dapprog dapprog.df
```

which compiles **dapprog.df** and generates the DOF file **dapprog**.

3.3.2 Compiling the FORTRAN host program

The FORTRAN host program is compiled using the f77 command:

```
f77  -o  hostprog  hostprog.f  -ldap
```

which compiles **hostprog.f** and generates the executable file **hostprog**. The program is listed in figure 3.3 below.

```
      program hostprog
      parameter (NROWS = 40, NCOLS = 70)
      integer in(NROWS,NCOLS)
      integer isum
      integer i, j, ires, dapcon
      common /indata/ in
      common /outdata/ isum
C
C Initialise input data
C
      do 10 j = 1, NCOLS
      do 10 i = 1, NROWS
      in(i,j) = i + j
10     continue
C
C Load DAP program
C
      ires = dapcon('dapprog')
      if (ires.ne.0) then
         write(0,1000) ires
1000        format(//'dapcon call failed (reason = ', i5, ')'/)
         stop
      endif
C
C Transfer input data to the DAP
C
      call dapsen('indata', in, NROWS*NCOLS)
C
C Execute DAP program
C
      call dapent('dapentry')
C
C Transfer result back from DAP and release DAP resource
C
      call daprec('outdata', isum, 1)
      call daprel
C
C Display result
C
      write(6,2000) isum
2000  format(//'Sum of matrix is: ', i8/)
      stop
      end
```

Figure 3.3 Example FORTRAN host program

3.3.3 Compiling the C host program

The C host program is compiled using the **cc** command:

```
cc  -o  hostprog  hostprog.c  -ldap
```

which compiles **hostprog.c** and generates the executable file **hostprog**. The program is listed in figure 3.4 on the page opposite.

3.3.4 Running the program

The host program is now executed in the usual way, by typing:

hostprog

The DAP program will be loaded into the DAP hardware (by **dapcon**) and data sent to the DAP (by **dapsen**). Control is then passed to the DAP (by **dapent**), and after the DAP processing is complete, control is returned to the host program and the result is returned to the host (by **daprec**). The DAP space used by this program is then released (by **daprel**), so that other users can access it, and the host post-processing starts. In this case the post-processing is simply a display of the result.

If **dapcon** fails to connect to the DAP it will return an error code, the host program will print the value of the code, and will stop. Appendix E on page 162 lists the meanings of the six possible **dapcon** return codes.

If you want to run the program on the simulator instead of on the hardware, you would first issue the **dapopt** command using the **-s1** flag:

```
dapopt  -s1  dapprog
```

For more details of **dapopt**, see section 3.4 below. The host program can then be run exactly as before, except that the DAP program will be executed by the simulator.

3.4 Specifying run-time options – dapopt

There are a variety of run-time options available when a DAP program is executed (for example, those options concerning **trace** output and diagnostic levels). The information on the options required is stored in the DOF file itself. You can modify the options by using the **dapopt** program.

You specify the options you want to change using **dapopt**'s flags. A single DOF filename has to be given as input, and by default the option information is added to that file. The **-o** flag lets you specify a new output file, the input file being left unchanged.

After you have used **dapopt**, the options you selected remain in force every time that DOF file is used, until explicitly changed by another run of **dapopt**; default values are used for those options which have not been modified by **dapopt**.

```
#include <stdio.h>

#define NROWS     40
#define NCOLS     70

extern int   dapcon();
extern void  dapent();
extern void  dapsen();
extern void  daprec();
extern void  daprel();

main()
{
   int in[NCOLS][NROWS];
   int isum;
   int i, j, ires;
/*
 * Initialise input data
 */
  for (j = 0; j < NCOLS; j++) {
       for (i = 0; i < NROWS; i++) {
            in[j][i] = (i + 1) + (j + 1);
       }
  }
/*
 * Load DAP program
 */
  ires = dapcon("dapprog");
  if (ires != 0) {
       (void) fprintf(stderr, "\n\ndapcon call failed (reason = %5d)\n\n", ires);
       exit(1);
  }
/*
 * Transfer input data to the DAP
 */
  dapsen("indata", &in[0][0], NROWS*NCOLS);
/*
 * Execute DAP program
 */
  dapent("dapentry");
/*
 * Transfer result back from DAP and release DAP resource
 */
  daprec("outdata", &isum, 1);
  daprel();
/*
 * Display result
 */
  (void) fprintf(stdout, "\n\nSum of matrix is: %8d\n\n", isum);
  exit(0);
}
```

Figure 3.4 Example C host program

Default values are used for all options in a DOF file if **dapopt** has never been run on the file (see section 3.4.6 on page 52 for details of the default values of the options).

For a summary of all the dapopt flags, see section 3.4.7 on page 52.

hardware or simulator

The **-s** flag specifies whether the program is to be run on the hardware or on the DAP simulator. The argument after **-s** should be 0 if you want to run the program on the hardware (the default), or 1 for the simulator.

The following command creates a DOF file called **sim** from the DOF file **d.out** which will run on the DAP simulator:

```
dapopt -s1 -o sim d.out
```

breakpoint edit mode

The **-b** flag to **dapopt** takes one argument, which can be 0 (the default) or 1. The effect of **-b1** is that whenever **dapent** is called from the host program, **psam** is entered before the first user instruction in the DAP program is executed. The main use of this **-b** option is to let you set breakpoints in your DAP program (or to execute the program in single-step mode right from the first user instruction in the program) without having to enter **psam** by some other means. For more information on **psam** and breakpoints, see chapter 4, which starts on page 57.

To enable breakpoint edit mode for DOF file **d.out**, you would issue the command:

```
dapopt -b1 d.out
```

3.4.1 trace execution control

The run-time trace level for FORTRAN-PLUS **trace** statements is defined by the **-f** flag to **dapopt**. The **trace** level is specified by its argument, which has to be in the range 0 to 5. **trace** statements are only executed if their level is less than or equal to that given by the **-f** flag. (As explained in section 2.4.3.1 on page 21, *compilation* of **trace** statements is itself controlled by the **-t** flag to **dapf**). The default value for **-f** is 5, that is, all compiled **trace** statements are executed.

Similarly, APAL **trace** statements are only executed if the argument to **dapopt**'s **-a** flag is greater than or equal to the level specified in the APAL **trace** statement. The **-a** flag argument should be in the range 0 to 15 (default 15). Assembly of APAL **trace** statements is controlled by the **-t** flag to **dapa** – see section 6.4.3 on page 123 for more details.

3.4.2 Run-time diagnostics

If a run-time error occurs in a DAP program, the diagnostic system will output information relating to the nature and location of the problem. The information displayed can be specified using the **-d** flag to **dapopt**, which takes an argument in the range 0 to 2 (default 0) and defines the output as:

Value of **-d** *argument*	*Effect*
0	A description of the error, its location (subprogram name and line number), and a route summary showing which procedures have been entered but not yet left. In the case of vector or matrix computational errors, an indication of which components were in error is also given .
1	As for 0, plus the names of and values held by all variables on the failing line.
2	As for 1, plus the names of and values held by all variables in the failing subprogram.

Note that information concerning line numbers and the values of variables is not generated if the subprogram was not compiled at the appropriate diagnostic level. See section 2.4.3 on page 21 for more details of run-time diagnostics.

diagnostics file

By default, diagnostic information is sent to the standard error stream. You can, however redirect it to a diagnostic file, which you can specify using **dapopt**'s **-D** flag. If **-D** is followed by a filename, then when you run the executable DAP program file and a run-time error occurs in the DAP, the diagnostics information will be written to the file specified in the **-D** option. Iif the file already exists its contents are deleted before the diagnostics are written to it. If you don't give any filename after **-D**, then the diagnostic output reverts to the standard error stream.

For example:

```
dapopt -D diag -d2 d.out
```

will, when **d.out** is being run, select diagnostic level 2 and redirect diagnostic output to the file **diag**.

Note that information written to the standard error stream by the host program is not redirected.

More details about diagnostic information are given in section 4.6 on page 75.

3.4.3 Run-time error action

After a DAP run-time error and the diagnostics described above have been output, the default action is to enter **psam**, the program state analysis mode subsystem, which is described in section 4.3 on page 60. You can change the default by using the **-e** flag with an appropriate argument, detailed at the top of the next page.

Value of **-e** *argument*	*Required action*
a	Abort the program and return immediately to the invoking host shell.
d	Dump the DAP status (including stack and variable information) to a file before returning to the shell.
p	Enter **psam**.
c	Ignore the error and try to continue execution.
cd or **dc**	Try to continue execution after dumping the DAP status to a file.

If you specify **d** or **dc**, the DAP state is output to a file with name *doffile*.**dr**, where *doffile* is the name of the DOF file being executed. You can examine this file later using **dapdb** (for more details see section 4.4 on page 72).

3.4.4 Examining options

The options selected by **dapopt** are stored in the DOF file itself. You can examine them for a particular DOF file by using the **-L** flag, which will give you the old and new values of all the options corresponding to the input and output DOF files.

For example:

```
dapopt -L -e d d.out
```

specifies that a dump is to be taken if a run-time error occurs in **d.out** and the options are to be listed to standard output.

An example of the output generated by the use of a **-L** flag is shown in figure 3.5 opposite. A command that might have generated such a display is:

```
dapopt -L -s1 -h 0X64 -t2 -f2 -e dc -S stats
```

3.4.5 Simulator options

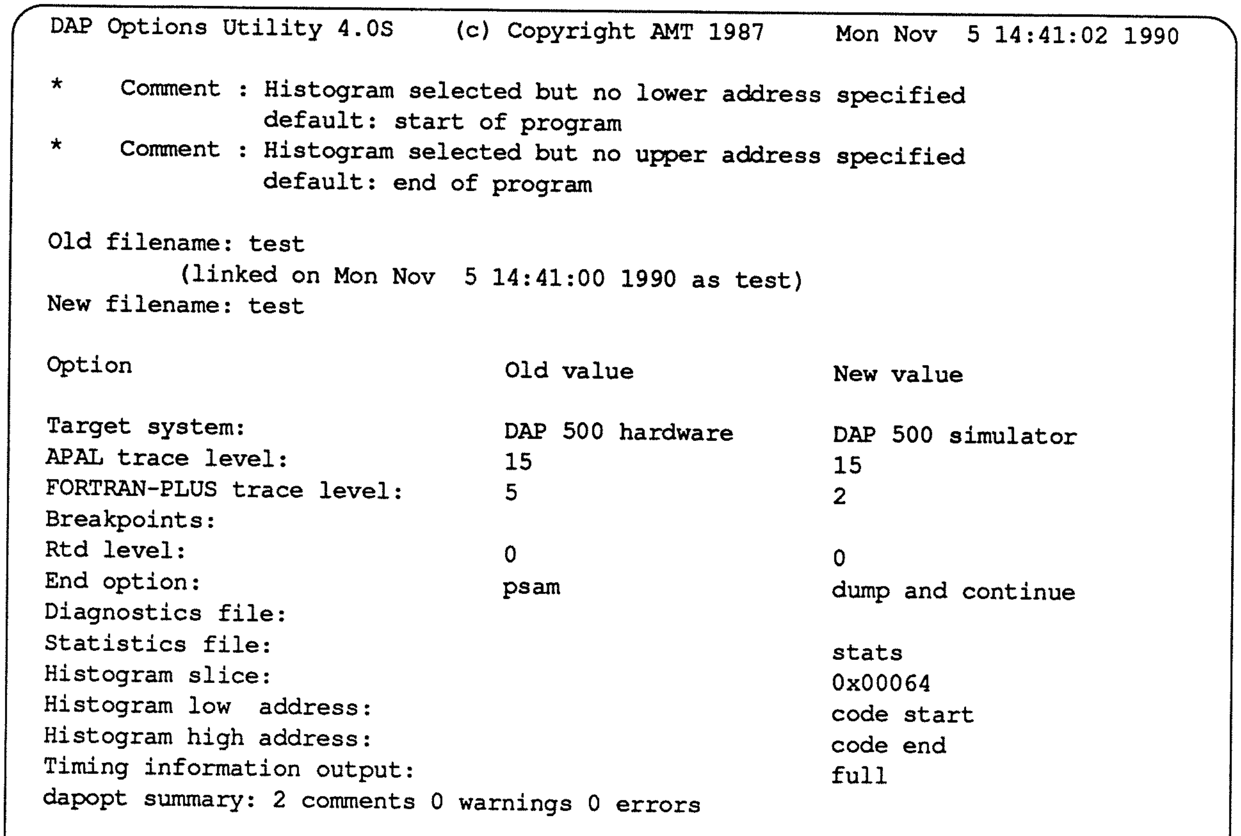
If the simulator option is effective (that is, if **-s1** has been specified) several extra facilities are available at run-time and these are selected by **dapopt** as described in this section.

If you specify any simulator-only options, but option **-s1** is not in force – perhaps because *no* **-s** option has been specified – these simulator-only options will still be recorded in the DOF file, and **dapopt** will output a comment.

3.4.5.1 Timing facilities

timing facilities on hardware and simulator

You can include appropriate DAP system calls in either FORTRAN-PLUS or APAL programs to measure the elapsed time and active running time on the DAP. You can also get timing information from within **psam**. For more details see [6], *DAP Series: DAP System Calls*, and for **psam**, section 4.7 on page 80. All these facilities are available on both hardware and simulator. There are additional timing facilities only available on the simulator.



```
DAP Options Utility 4.0S    (c) Copyright AMT 1987     Mon Nov  5 14:41:02 1990

*    Comment : Histogram selected but no lower address specified
               default: start of program
*    Comment : Histogram selected but no upper address specified
               default: end of program

Old filename: test
        (linked on Mon Nov  5 14:41:00 1990 as test)
New filename: test

Option                          Old value              New value

Target system:                  DAP 500 hardware       DAP 500 simulator
APAL trace level:               15                     15
FORTRAN-PLUS trace level:       5                      2
Breakpoints:
Rtd level:                      0                      0
End option:                     psam                   dump and continue
Diagnostics file:
Statistics file:                                       stats
Histogram slice:                                       0x00064
Histogram low  address:                                code start
Histogram high address:                                code end
Timing information output:                             full
dapopt summary: 2 comments 0 warnings 0 errors
```

Figure 3.5 An example of a **dapopt** *listing*

timing facilities on simulator only

If you run your program on the simulator you can get an estimate of the time the program would take to run on the DAP hardware in units of machine clock cycles. The period of one clock cycle depends on the version of DAP on which you intend to run your code.

You can ask for timing for the complete run, and at standard or user-defined points within the DAP program. The **-t** flag specifies the timing option required by its argument:

0 No timing information

1 Standard timing information

2 Full timing information

standard timing

If you ask for standard timing the estimate for the run as a whole is printed, together with a line recording the total number of instructions executed.

full timing

Full timing outputs the same information as standard timing, with the addition of intermediate reports during the run whenever a defined event occurs. The events which cause a report to be output are listed at the top of the next page.

Event *Details of the event*

EXIT This event corresponds to an **exit** instruction in APAL, or a **return** statement in FORTRAN-PLUS.

JSL This event corresponds to a **jsl** instruction in APAL, or a **call** statement or a function reference in FORTRAN-PLUS.

SVC This event corresponds to a system supervisor call, and is used, for example, by the trace and run-time diagnostics facilities. An **svc** is also one of the last events before any DAP program ends.

END OF RUN

The format of the information output depends on the type of the event, but in general is as follows:

```
Event: type at PC = X n
Time since last event = x beats
Time for run so far = y beats
```

where *type* is one of the events above, and *n* is the instruction address (in hexadecimal) at which the event occurred. The times are given in *beats* or machine clock cycles.

user-defined timing requests

For both standard and full timing a report similar to an event report is output whenever a FORTRAN-PLUS or APAL:

```
pause 9999
```

statement is encountered.

These **pause 9999** statements only have this special timing significance when you are using the simulator and have specified one or more of the timing options. In all other cases **pause 9999**s are treated as normal user-defined **pause** statements and the program is suspended in the normal way.

The information output at a user-defined timing point is:

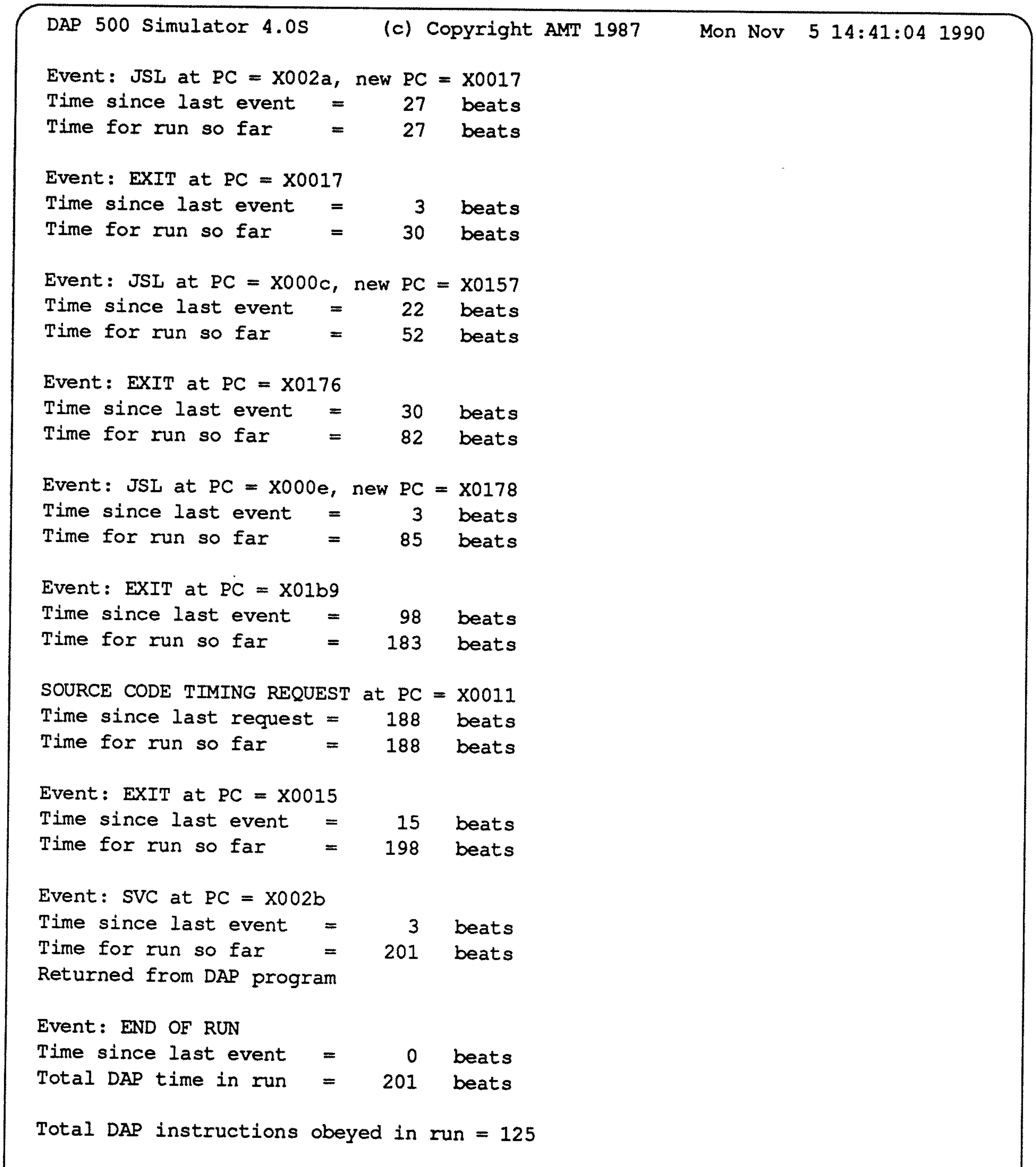
```
SOURCE CODE TIMING REQUEST at PC = x n
Time since last request = x beats
Time for run so far = y beats
```

Note that the system code associated with user-defined timing points keeps an estimate of the time since it encountered the previous user-defined timing point; that record is independent of the record it keeps of the full timing events described above.

An example of a full timing listing incorporating a user-defined timing request is given in figure 3.6 opposite.

using timing information

You can convert timing estimates specified in terms of beats (machine clock cycles) to actual execution times on DAP hardware by multiplying the estimates by the clock cycle time of the hardware being simulated. Where a substantial part of the processing occurs in the DAP program, and there is little



```
DAP 500 Simulator 4.0S      (c) Copyright AMT 1987      Mon Nov  5 14:41:04 1990

Event: JSL at PC = X002a, new PC = X0017
Time since last event    =    27   beats
Time for run so far      =    27   beats

Event: EXIT at PC = X0017
Time since last event    =     3   beats
Time for run so far      =    30   beats

Event: JSL at PC = X000c, new PC = X0157
Time since last event    =    22   beats
Time for run so far      =    52   beats

Event: EXIT at PC = X0176
Time since last event    =    30   beats
Time for run so far      =    82   beats

Event: JSL at PC = X000e, new PC = X0178
Time since last event    =     3   beats
Time for run so far      =    85   beats

Event: EXIT at PC = X01b9
Time since last event    =    98   beats
Time for run so far      =   183   beats

SOURCE CODE TIMING REQUEST at PC = X0011
Time since last request =   188   beats
Time for run so far      =   188   beats

Event: EXIT at PC = X0015
Time since last event    =    15   beats
Time for run so far      =   198   beats

Event: SVC at PC = X002b
Time since last event    =     3   beats
Time for run so far      =   201   beats
Returned from DAP program

Event: END OF RUN
Time since last event    =     0   beats
Total DAP time in run    =   201   beats

Total DAP instructions obeyed in run = 125
```

Figure 3.6 An example of a full timing listing

interaction with the host, these execution times will be close to the active run time when you are running the program on a DAP. If a large part of the processing is being carried out on the host, or if a lot of data is transmitted between the host and

DAP during execution, then the total active running time is dominated by the scheduler on the host processor.

3.4.5.2 Program profiling and execution histogram

execution profile on hardware and simulator

A FORTRAN-PLUS program that has been compiled with the **`-p`** option to **`dapf`** creates a high-level execution profile when run; the facility is available on both hardware and simulator. A similar facility is available for APAL programs. You can analyse the profile using **`dapprof`** – see section 3.5 on page 54 for more details.

There is a low-level profiling facility for both FORTRAN-PLUS and APAL programs which is only available on the simulator. The rest of this section 3.4.5.2 describes that facility.

execution histogram on simulator only

You use **`dapopt`**'s **`-h`** option to ask for an execution histogram of the machine instructions in a DAP program run on the simulator. You give an argument *n* to the option, where *n* specifies that you would like the instructions in the program divided into 'slices' of *n* instructions each. The system then keeps a count for each slice of how many times any of the instructions from that slice is executed during the program. The histogram is sent to standard output. If you want to disable histogram output, set *n* to 0; that is, specify the option **`-h0`**.

Each line of the histogram records the instruction range to which the line refers, and displays the total instruction count in that slice during the run, both as a number and pictorially in the form of one or more asterisks. At the start of the histogram a line records the number of instructions represented by a single asterisk.

instruction addresses

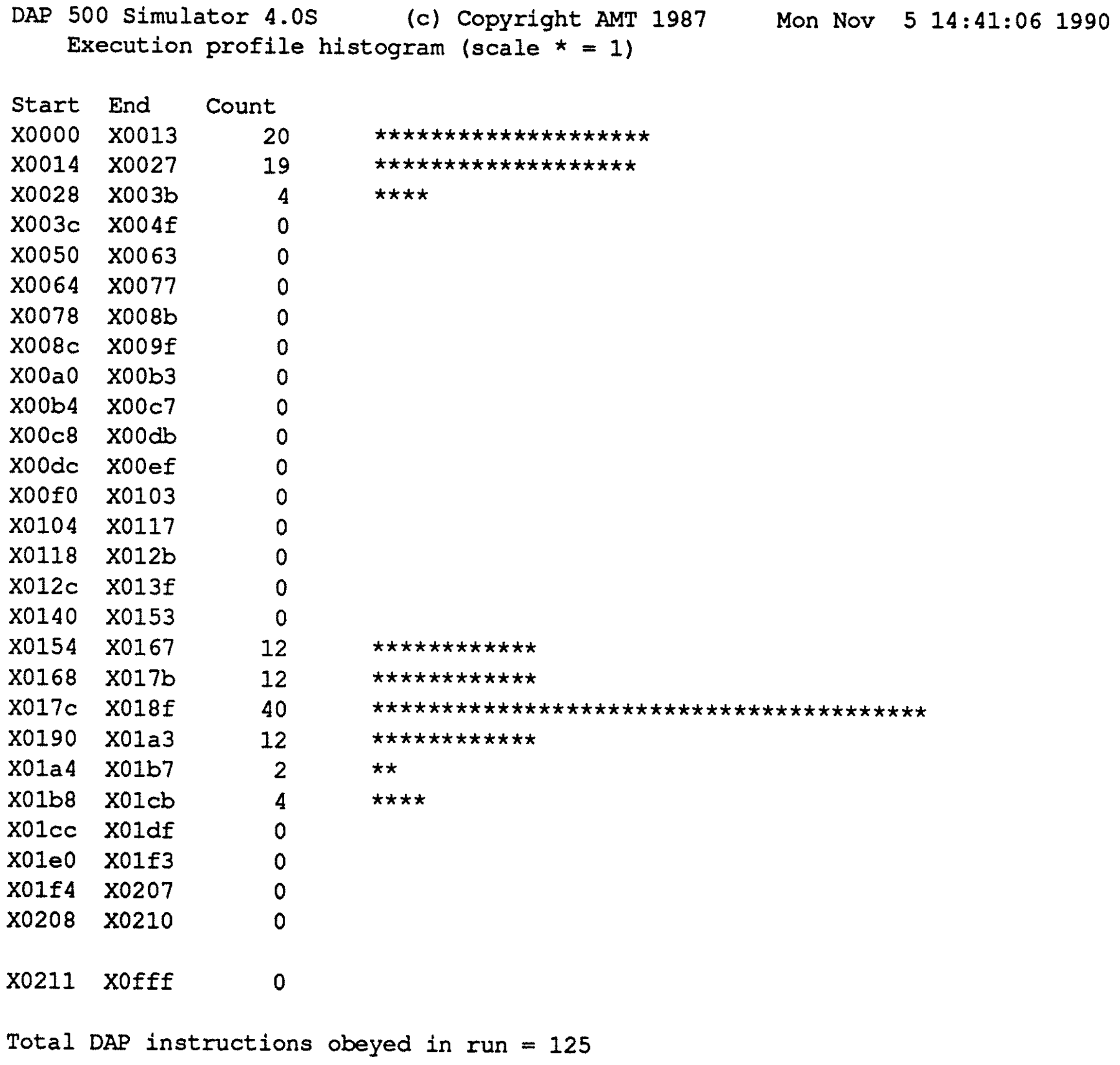
You can relate the instruction addresses in the histogram to user-written or AMT-supplied procedures by consulting a full map produced by the consolidator. You get this map by specifying **`-m3`** to **`dapa`** (for more details, see section 6.7 on page 124) or **`-m3`** to **`dapf`** (see section 2.7 on page 32). The information contained in the consolidator map is described in section 2.5.2 on page 25. By using an assembler listing, APAL programmers can relate instruction addresses to specific instructions in their code sections. FORTRAN-PLUS programmers can only associate an instruction range with a complete procedure; you can't usually associate a given instruction slice with only some of the lines in a procedure.

If you specify **`-h`***n*, then by default, the histogram is a profile of the whole of the DAP program. You can specifiy alternative start or end addresses with the **`-l`***n* flag (for start, or lower address) and the **`-u`***n* flag (for end, or upper address), where *n* is the address. If you use one or both of these flags, then extra line(s) in the histogram record the total number of instructions executed in the address range(s) not covered in the rest of the histogram.

address in octal, decimal, or hex

You can specify the address *n* after the **-l** or **-u** flags in decimal, octal or hexadecimal. The system assumes that numbers starting with 0X (that is, zero-X) are in hexadecimal; those starting with 0 (that is, zero) are in octal, and all others are in decimal.

When you use the histogram facility, a final line records the total number of instructions obeyed in the run. An example of a histogram is given in figure 3.7 below. The **dapopt** option to produce this histogram could have been **-h0X14**.



```
DAP 500 Simulator 4.0S      (c) Copyright AMT 1987     Mon Nov  5 14:41:06 1990
    Execution profile histogram (scale * = 1)

Start  End     Count
X0000  X0013     20      ********************
X0014  X0027     19      *******************
X003b  X003b      4      ****
X003c  X004f      0
X0050  X0063      0
X0064  X0077      0
X0078  X008b      0
X008c  X009f      0
X00a0  X00b3      0
X00b4  X00c7      0
X00c8  X00db      0
X00dc  X00ef      0
X00f0  X0103      0
X0104  X0117      0
X0118  X012b      0
X012c  X013f      0
X0140  X0153      0
X0154  X0167     12      ************
X0168  X017b     12      ************
X017c  X018f     40      ****************************************
X0190  X01a3     12      ************
X01a4  X01b7      2      **
X01b8  X01cb      4      ****
X01cc  X01df      0
X01e0  X01f3      0
X01f4  X0207      0
X0208  X0210      0

X0211  X0fff      0

Total DAP instructions obeyed in run = 125
```

Figure 3.7 An example of a histogram

3.4.5.3 *Specifying a statistics file*

The timing and histogram information described above is sent to standard output by default. However, you can re-direct it to a file by using the `-S` flag. This takes one argument, the name of the file to be written. If the file already exists, its contents will be deleted when the DAP program is next executed before statistics are written to the file. If you use the `-S` flag without an argument the default is restored; that is, statistics are sent to standard output.

3.4.6 Restoring default options

The `-x` flag restores all run-time options to their default values. It also cancels any preceding flags specified in the current **`dapopt`** command. However, if you specify any flags after a `-x` the flags are acted upon in the usual way.

For example:

```
dapopt -x -f 2 d.out
```

will first reset all options in the DOF file `d.out` and then set the run-time FORTRAN-PLUS `trace` level to 2, whereas:

```
dapopt -f 2 -x d.out
```

would leave `d.out` with all the default options set (including the run-time FORTRAN-PLUS `trace` level as 5), the `-f` flag being ignored as it comes before the `-x` flag.

When a DOF file is first created, its run-time options take default values equivalent to applying the **`dapopt`** command with the following flags:

```
-a 15 -b0 -d 0 -D -e p -f 5 -S -s 0
```

3.4.7 `dapopt` flags

This section contains a summary of all the **`dapopt`** flags.

`-a`*n* Set the maximum level of APAL `trace` statements to be executed to *n*, where *n* is in the range 0 to 15.

The default is *n* = 15.

`-b`*n* Take the specified action whenever **`dapent`** is called from the host program, where *n* is one of:

0 Enter the DAP program.

1 Pass control directly to **`psam`**, and do not enter the DAP program.

The default is *n* = 0.

`-d`*n* Set diagnostics level *n* to 0, 1, or 2.

The default is *n* = 0.

`-D` *name* Send diagnostics to file *name*.

`-D` Send diagnostics to the standard error stream (the default).

-e*x* Take the specified action if a run-time error occurs or a **pause** is executed, where *x* is one of:

- **a** Abort.
- **c** Continue.
- **p** Enter **psam** .
- **d** Dump.
- **dc** or **cd** Dump and continue.

The default is *x* = **p** .

-f*n* Set the maximum level of FORTRAN-PLUS **trace** statements to be executed to *n*, where *n* is in the range 0 to 5.

The default is *n* = 5.

-h*n* Generate a histogram based on slices of *n* instructions (simulator only). A value of 0 generates no histogram.

The default is *n* = 0.

-l*n* Set the histogram lower limit to code address *n* (simulator only).

The default is start of program.

-L List the file options to standard output.

-o *name* Put the DOF output in file *name* .

-q Suppress **dapopt** comments output.

-S *name* Send statistics to file *name* (simulator only).

-S Send statistics to standard output (the default).

-s*n* Run the DAP program as specified, where *n* is one of:

- **0** Run on the DAP hardware.
- **1** Run on the DAP simulator.

The default is *n* = 0.

-t*n* Provide timing information as specified by *n* (simulator only):

- **0** No timing information.
- **1** Standard timing information.
- **2** Full timing information.

The default is *n* = 0.

-u*n* Set the histogram upper limit to code address *n* (simulator only).

-x Reset default values for all options – ignore previous flags (if any) in this call to **dapopt**.

-y Suppress the output of a DOF file.

3.5 Using the high-level execution profiler

3.5.1 Introduction

One of the problems associated with increasing the speed of execution of a DAP program is to find out what percentages of total execution time are spent in what parts of the program. Once you have this information, you can optimise code that is critical to program speed-up.

low-level execution histogram on simulator

When you run a FORTRAN-PLUS or APAL program on the simulator, a low-level execution histogram of the program is available through **`dapopt`**'s **`-h`** flag and can help you to optimise your code. Section 3.4.5.2 on page 50 gives more details of this simulator-only execution profile.

high-level execution profiler on hardware or simulator

for a FORTRAN-PLUS program

When you run a FORTRAN-PLUS program on DAP hardware or simulator, if you have specified the **`-p`** flag to **`dapf`** when you compiled the program, then an execution profile of that program is written to file **`dmon.out`** in your current directory. You can then examine **`dmon.out`** using the utility **`dapprof`**, described later in this section.

and an APAL program

A similar facility is available for APAL programs. You need to specify the **`-p`** flag to **`dapa`**, but you also need to **`#include`** various AMT system macros held in file **`amtmacs.da`** in your APAL source. If your APAL program makes use of the AMT system macros, you might already have **`#included`** **`amtmacs.da`** – or you might have the line:

```
#include usrmacs.da
```

at appropriate places in your APAL source. AMT upgraded the macros in **`usrmacs.da`**, and re-issued them in file **`amtmacs.da`**, and it is the upgraded macros that the high level profiler needs when APAL code is being run.

Hence, if you are already using **`usrmacs.da`** in your APAL source, if you change filename **`usrmacs.da`** to **`amtmacs.da`**, then you can make use of the high-level execution profiler with your APAL code.

If your APAL program does not use the various AMT system macros, but you want to use the execution profiler when you run your APAL code, you will need to include the line:

```
#include amtmacs.da
```

in every APAL module for which you want execution profile information. You will also need to adopt the entry and exit conventions described in [3], *DAP Series: APAL Langauge.*

When you run your APAL program, the profiling information is written to file **`dmon.out`** in your current directory.

for a mixed FORTRAN-PLUS and APAL program

If your program contains FORTRAN-PLUS and APAL code sections, then provided you compile or assemble the relevant sections as described above, when you run the program **`dmon.out`** will contain the profile for the whole program.

recompile without the -p flag when development is over

Once you have finished program development, AMT recommends that you recompile your program, having removed the **-p** flag – and other flags associated with development tools – from the **dapf** or **dapa** command line.

3.5.2 Analysing the profile with dapprof

To analyse the execution profile, run the command:

```
dapprof dof-file-name
```

where *dof-file-name* is the name of the DOF file containing the DAP code associated with the profile.

You will then see a display similar to:

```
name       #calls    %cycles   cycles    %s-cycles  s-cycles  s-cycles/call
AAA             1      97.88    16809         2.69       452         452.00
S               1      95.25    16357        56.02      9417        9417.00
BG              1      40.41     6940        11.98      2013        2013.00
U               1      28.69     4927        29.31      4927        4927.00
```

where the column headings have the meanings detailed below:

name The name of a routine in the DAP program under scrutiny.

#calls The number of times this routine is called during the execution of the program.

%cycles The percentage of all machine cycles spent in this routine, including cycles spent in routines called by this routine.

cycles The total number of machine cycles spent in all calls to this routine, including cycles spent in routines called by this routine.

%s-cycles The percentage of all 'self-cycles' spent in this routine – that is, excluding machine cycles spent in routines called by this routine.

s-cycles The total number of all 'self-cycles' spent in this routine – that is, excluding machine cycles spent in routines called by this routine.

s-cycles/call The number of 'self-cycles' spent in this routine per call to the routine – that is, excluding machine cycles spent in routines called by this routine.

The output from **dapprof** is sorted in descending numerical order on the **%cycles** field by default. You can specify an output sorted in descending numerical order on the **s-cycles** field by specifying the **-s** flag to **dapprof**, as perhaps:

```
dapprof -s progtest
```

figures include overeheads

Note that system overheads in calling routines – and returning from them – are included in the above figures, but the time taken by the profiler itself to extract and process the profiling information is not included in the figures.

Chapter 4

Program testing

4.1 Introduction

This chapter describes the facilities available to you for testing your DAP programs, whether written in FORTRAN-PLUS or APAL code, or in both.

4.2 Overview of program testing

When a DAP program is running, various program events can suspend execution and pass control back to the DAP run-time diagnostic system running on the host. These events include, amongst others, DAP run-time errors and **pause** statements in your DAP program.

The run-time diagnostic system then outputs a suitable report, and takes whatever action is specified in your most recent invocation of **dapopt**, the DAP run-time options program. By default, control passes to **psam**, the program state analysis mode sub-system (the on-line debugger). For more details of **dapopt**, see section 3.4 on page 42.

One of the options available in **dapopt**, and one of the commands in **psam**, lets you take a dump of the DAP state – the array store part of your DAP program block. You can then examine the dump, using **dapdb**, a post-mortem dump analysis program similar in function to **psam** .

This chapter describe these on-line debugging and post-mortem analysis facilities in detail. It also describes other DAP diagnostic facilities.

4.2.1 On-line facilities

psam, the interactive debugger, is a major diagnostic tool. It is entered by default when execution of your DAP program stops prematurely, and control passes to the run-time diagnostic system.

Once in **psam**, you can look at your source code and the values of variables in your code, and display the contents of your part of the array store. You can insert breakpoints in the code, single-step your way through the code, or continue normal program execution. These facilities that allow you to single-step or continue execution of your program actually transfer control back from **psam** to the run-time support system. Execution then continues for one or more instructions or until a **pause**, breakpoint or run-time error is reached, after which control returns to **psam** . If there are no **pauses**, breakpoints or errors in the rest of the program, execution will continue to the end of the program, and control will return to the host command line in the normal way.

The **pause** or breakpoint that returns control to **psam** can be the next encountered, or next but one or more – that is, control passing over a mixture of (*n*–1) **pauses** and breakpoints.

A macro facility lets you execute **psam** commands from a file. You can also take dumps of the DAP state for later use by the post-mortem dump analyser, **dapdb** .

psam, with its breakpoints facility, offers the same control over program flow that is offered by the **pause** statement, with the advantage that program re-compilation or re-assembly is not necessary.

Another diagnostic facility is provided by **trace**. **trace** statements embedded in your source code will output the values of nominated variables during program execution, but without passing control to **psam** . **trace** facilities are much simpler than those available in **psam**, but they do let you test a DAP program in batch mode, sending any diagnostic output to a file as execution proceeds.

a psam session

'A **psam** session' is a convenient expression to describe a session in which you use **psam** to debug your DAP program on-line; the expression – or just 'session' – is used in this chapter to describe this type of interactive DAP program debugging.

4.2.2 Post-mortem facilities

Most of the functionality available in **psam** is also available in the post-mortem dump analyser, **dapdb**. Although you cannot execute any code from within **dapdb**, you can look at your source code and the values of variables in array store.

4.2.3 Summary of psam and dapdb commands

Listed below are all the commands available in either **psam** or **dapdb**; a few commands are only available in one or other and are noted accordingly.

Command	*Its functionality*
alias	Creates alternative name(s) for **psam** or **dapdb** command(s).
array	Displays the contents of an area of array store.
attributes	Displays the attributes of variable(s).
backtrack	Displays details of the procedure(s) currently on the stack.
breakpoints	Displays the current breakpoint settings (**psam** only).
clear	Clears breakpoint(s) (**psam** only).
code	Disassembles and displays APAL object code from the current code section (**psam** only).
continue	Continues execution of the DAP program (**psam** only), possibly ignoring a number of **pauses** and breakpoints.
core	Changes the current dump file to the one specified (**dapdb** only).

Command	*Its functionality*
date	Displays the current time and date.
disable	Disables breakpoint(s) (**psam** only).
display	Sets up a list of FORTRAN-PLUS variable(s) whose contents are to be displayed on entry to **psam** .
down	Changes the current procedure to that procedure which is next lower on the stack.
dump	Dumps the current DAP state to a file (**psam** only).
echo	Displays its own arguments.
enable	Enables breakpoint(s) (**psam** only).
errors	Displays the positions of FORTRAN-PLUS computational errors.
file	Changes the current file to the one specified.
help	Displays help information on **psam** and **dapdb** commands.
history	Displays the commands used earlier in a **psam** or **dapdb** session.
list	Lists from the current file.
macro	Executes **psam** or **dapdb** commands from a file.
map	Displays map(s) of your program's occupancy of code store (MCU or co-processor) or array store, or some combination of all three.
masks	Displays user-defined error interrupt masks.
message	Repeats the information displayed on entry to **psam** or **dapdb** .
next	Steps program execution through one or more FORTRAN-PLUS source statements, starting with the next statement and treating any procedure calls as single statements (**psam** only).
print	Displays the contents of the specified FORTRAN-PLUS variable(s).
procedure	Changes the current procedure to that specified.
quit	Quits a **psam** or **dapdb** session.
registers	Displays any or all of the MCU, edge and PE registers, the carry and overflow flags in APAL, and the hardware DO loop iteration number.
save	Saves the current settings of **psam** or **dapdb** environment variables to file **.defaults** in your home directory.
select	Changes the current DAP state dump to the one specified (**dapdb** only).
set	Sets **psam** or **dapdb** environment variable(s).
status	Displays the current breakpoints in command format (**psam** only).
step	Steps program execution through one or more FORTRAN-PLUS source statements, starting with the next statement and treating each statement in a procedure call as one statement (**psam** only).
stepi	Steps program execution to the next APAL instruction (**psam** only).
stop at	Sets a breakpoint at the start of a FORTRAN-PLUS source statement (**psam** only).
stop in	Sets a breakpoint on the first executable line of a FORTRAN-PLUS procedure (**psam** only).
stopi at	Sets a breakpoint at a given offset in an APAL procedure (**psam** only).
stopi in	Sets a breakpoint at the start of an APAL procedure (**psam** only).
time	Displays total execution time, and time since last **time** command was issued.
top	Changes the current procedure to the procedure at the top of the stack.
unalias	Deletes alternative name(s) for **psam** or **dapdb** commands.
undisplay	Clears the list of variables to be displayed on entry to **psam** .
unset	Unsets the values of **psam** or **dapdb** environment variables.
up	Changes the current procedure to that procedure which is next higher in the stack.

See section 4.3 below for more details of the **psam** commands; see section 4.4 on page 72 for more details of

dapdb. See section 4.7, starting on page 80, for the formal specification of **psam** and **dapdb** commands.

4.3 Program state analysis mode (psam)

4.3.1 Introduction

When a run-time error or similar interrupt occurs in a DAP program, the system generates a diagnostic report (described in detail in section 4.6 on page 75) and control then passes to the run-time diagnostic system. What happens next depends on the parameter specified for the **-e** option in **dapopt** (see section 3.4.3 on page 45 for more details). If **psam** has been selected (the default), then program state analysis mode is entered, and you are presented on your host screen with the **psam** prompt:

```
psam:
```

files used in psam examples

In the discussion that follows in this chapter, the displays output by the various **psam** commands are shown, all generated as a result of exploring in **psam** a simple DAP program. The program consists of a short entry subroutine (**entdap**) held in file **esdap.df**. **entdap** calls a function (**add3**), which is held in file **fadd3.df**. The simple host program (**exhost**) held in file **hostex.f** is also shown. The listings of the three files are shown in figure 4.1 opposite:

The two FORTRAN-PLUS files were compiled with the command:

```
dapf -g -o dapobj esdap.df fadd3.df
```

The host FORTRAN file was compiled using the command:

```
f77 -o hostobj hostex.f -ldap
```

The **dapopt** utility was run with the command:

```
dapopt -b1 -s1 dapobj
```

For more details of the **dapopt** flags, see section 3.4.7 on page 52.

DAP program execution was initiated on the simulator in the normal way, with the command:

```
hostobj
```

breakpoint edit mode

One of **dapopt**'s options is **-b**. If you specify **-b1**, then every time the DAP program is entered (that is, every time **dapent** is called from your associated host program), control passes directly to **psam**, in *breakpoint edit mode*, without execution starting.

Having run **dapopt** with flag **-b1**, when you start DAP program execution (with **hostobj** in our example) you would get the display shown in figure 4.2 opposite.

```
1       entry subroutine entdap
2
3       integer mvar(*2,*3),msum
4       external function add3
5       integer add3(*,*)
6       mvar=4
7       msum=sum(mvar*add3(mvar))
8       pause 2
9       return
10      end
```

FORTRAN-PLUS entry subroutine **entdap** *in source file* **esdap.df**, *and in DOF file* **dapobj**

```
1       function add3(im)
2
3       integer add3(*size(im,1),*size(im,2)),im(*,*)
4       add3=(im+3)
5       pause 1
6       return
7       end
```

FORTRAN-PLUS function **add3** *in source file* **fadd3.df**, *and also in DOF file* **dapobj**

```
1       program exhost
2
3       integer ires, dapcon
4       ires = dapcon('dapobj')
5       if (ires .eq. 0) then
6         call dapent('entdap')
7         call daprel
8       endif
9       stop
10      end
```

FORTRAN host program **exhost** *in source file* **hostex.f**, *and object file* **hostobj**

Figure 4.1 Example DAP and host source files used in **psam** *example displays*

Like all **dapopt** options, the value of the **-b** option is held in the DOF file, so it keeps its value from session to session and until you change it in another call to **dapopt**. Having specified **-b1** in one call to **dapopt**, if you later specify **-b0** in another call to **dapopt**, subsequent DAP program execution will start immediately **dapent** is called. The default is **-b0**.

```
host% hostobj
Entering Breakpoint Edit Mode
FORTRAN-PLUS Subroutine ENTDAP at Line 6 in File esdap.df

File ./esdap.df
  6>   :     mvar=4
End of Report
psam:
```

Figure 4.2 **psam** *display when* **dapopt***'s* **-b1** *flag is in force, and program execution has just started*

If you are in breakpoint edit mode, execution of the DAP program has not yet started, but you can insert *breakpoints* in your DAP program. The full set of **psam** commands are supported in breakpoint edit mode, although the output you get from some commands might not be helpful, as program execution has not started!

displaying output

One of the features of **psam** is that you can re-direct the output generated by any **psam** command from the screen (the default) to a nominated file.

Hence:

```
print var > results
```

will re-direct the contents of the variable **var** to the file **results**, and:

```
stop at 25 > details
```

will set up a breakpoint at line 25 in the current FORTRAN-PLUS source file, and will re-direct the message (describing the breakpoint the command has just set up) from the screen to file **details**. The verb 'display' will normally be used in this chapter to mean that the output is usually displayed on the screen, but can be re-directed to a nominated file.

Note: There is a **display** command in **psam**, that doesn't display anything! It sets up or changes a list of variables to be displayed on the next entry to **psam**. See page 85 for more details.

psam – a window

psam is essentially a window on the state of your DAP program, and its associated files; a window that you can move around at will.

in time ...

Because **psam** lets you set breakpoints in your DAP program, it lets you suspend execution at any point in your code, and examine a 'snapshot' of the DAP state. Having examined the DAP state, you can choose either to **quit** the DAP program, or to **continue** (restart) it, in which case **psam** will be re-entered if and when control is transferred back to the run-time diagnostic system. You can also **dump** the DAP state to a file for later analysis by **dapdb**.

As an alternative to **continue**, commands **step** and **stepi** let you restart your DAP program, and then automatically re-enter **psam** after a specified number of FORTRAN-PLUS statements or APAL instructions have been executed, at which point you can then examine the new DAP state.

... and in store

psam's current file, line, procedure and instruction

When **psam** is entered, **psam**'s *current procedure* is the procedure holding the FORTRAN-PLUS statement that was currently executing or the last APAL instruction to be executed. (The term *procedure* covers any FORTRAN-PLUS subroutine or function, or APAL code section.) The file holding the FORTRAN-PLUS source code is **psam**'s *current file*, and the line holding the statement that was currently executing is the *current line*. (The information **psam** relies on to label a file as current is only kept if the program was compiled with the **-D** option to **dapf** set to 1 or more – the default is 2.) For APAL procedures, when **psam** is entered, its *current instruction* is the instruction that will be executed if the program is restarted.

psam's active procedures

A DAP program usually contains many procedures. When program execution is suspended, control might have entered several procedures, but not yet left them; these are **psam**'s *active* procedures.

Many of **psam**'s facilities operate on the current file, line, procedure or instruction, and **psam** commands let you change these from the ones applicable when execution was suspended, to others. **psam** lets you examine all variables in active procedures, but only has information on common or static variables for non-active procedures.

If execution resumes and subsequently **psam** is entered again, the active procedures and the current file, line, procedure and instruction are those relevant to the new DAP state.

psam does not affect program outcome

psam is a window on the DAP state, and although its breakpoints and single-stepping let you control how your program runs, it does not affect any program data, and will not affect the final outcome of the program – unless you type **quit** .

All the features available to you in **psam** are described briefly below; full details are given in section 4.7, starting on page 80.

4.3.2 Interface with dapdb

psam is an on-line debugger. You use **dapdb** off-line: from within **psam** you can **dump** the DAP state (the contents of the array store) to a file for later examination by **dapdb**.

4.3.3 Examining variables

psam lets you **print** any FORTRAN-PLUS common or static variables, and FORTRAN-PLUS local variables in active procedures.

In the example program listed on page 61, if you have issued a **step 3** command after the program state shown in figure 4.2 on page 61, you would see the display at the top of the next page.

```
psam: step 3
Stepped to FORTRAN-PLUS Function ADD3 at line 5 in File fadd3.df

File ./fadd3.df
  5>   :        pause 1
psam:
```

Figure 4.3 **psam** *display after a* **step** *command has been issued*

If you then issued the command **print im** you would get the display:

```
psam: print im
Integer Matrix Parameter IM in 32 bits -
        dimensions: (*2,*3)

(1:2,1:3)              4 (* 6)

psam:
```

Figure 4.4 **psam** *display when all components of a matrix variable are* **print***ed*

For variables with more than one component, you can display all or only some of the components of the variables. Hence you could examine the values held in a complete array of matrix variables, or in just one component of a single vector. For example, you could print out the single component **im(1,2)** with the command **print im(1,2)**, and get the display:

```
psam: print im(1,2)
Integer Matrix Parameter IM in 32 bits -
        dimensions: (*2,*3)

(  1,  2)             4

psam:
```

Figure 4.5 **psam** *display when one component of a matrix variable is* **print***ed*

You can display the **attributes** of a variable (its type, mode, shape, size, address, and so on), in **im**'s case with the command **attributes im**, and get the display shown at the top of the next page.

You can use attribute information in conjunction with the **array** command to examine the contents of array store where FORTRAN-PLUS variables are located.

The **display** command sets up (or adds to) a list of variables whose contents are to be output every time **psam** is entered; the command **undisplay** removes all variables from that list.

```
psam: attributes im
Integer Matrix Parameter IM in 32 bits -
        dimensions: (*2,*3)
        addressed by Pointer on Stack at offset 0..64
        current address of Pointer: 194.0.0
        current contents of Pointer: 160.0.0

psam:
```

Figure 4.6 **psam** *display of the* **attributes** *of a variable*

The **errors** command displays the positions of FORTRAN-PLUS computational errors; **masks** displays any user-defined error interrupt masks that are current.

4.3.4 Breakpoints

psam lets you set breakpoints in your FORTRAN-PLUS and APAL programs. So, while you are in **psam** you can specify points at which subsequent execution of the code is to be suspended. These points can be either at the start of a specified procedure, or at a specified statement or instruction in the code. The breakpoints are lost when you end your **psam** session, although you can save them to a file for future use.

For example, if you were at the **psam** prompt in the example program **dapobj**, and you issued:

```
stop in add3
```

you would then get the display:

```
psam: stop in add3
Ref: File Name          Line  Procedure         Ofst  Activity Command
  1: fadd3.df           4     ADD3              #27    enabled
psam:
```

Figure 4.7 **psam** *display when a breakpoint is set up*

In a FORTRAN-PLUS program, **stop in** sets a breakpoint at the first executable statement of a specified procedure, **stop at** sets a breakpoint on a specified line in the current file. Commands **stopi at** and **stopi in** have a similar effect in an APAL program: **stopi in** sets a breakpoint at offset 1 (the normal entry point) in a procedure; **stopi at** sets a breakpoint at a given offset in the current procedure. Once **psam** has accepted a **stop** or **stopi** command, it issues a breakpoint reference number, which you can use later in the current **psam** session to refer to that breakpoint.

A useful feature in **psam** is that you can attach a command to a breakpoint specification, perhaps as:

```
stop in test "print a* >> results"
```

which would insert a breakpoint at the start of the FORTRAN-PLUS procedure **test**. When the breakpoint is reached, execution is suspended and the command **print a* >> results** is executed. (See section 4.3.11 on page 71 for a discussion of why you need quotes in the **stopi** command.)

You can **disable**, **enable** and **clear** existing breakpoints. The command **breakpoints** on its own displays the current breakpoints. **status** displays the breakpoint information in command format; you can redirect the **status** output to a file to save the information, so that you can use it in a later session.

In our example program, you could disable the breakpoint you set earlier with:

```
disable 1
```

If you then wanted to set another breakpoint at line 6 say in the current procedure **add3**, you would issue:

```
stop at 6
```

and you would get a display similar to figure 4.7 above. If you now issued the **breakpoints** command you would get the display:

```
psam: breakpoints
Ref: File Name       Line  Procedure         Ofst  Activity Command
  1: fadd3.df        4     ADD3              #27    disabled
  2: fadd3.df        6     ADD3              #7e   enabled
psam:
```

Figure 4.8 Display of the breakpoints in a **psam** *session*

Note that you can add to or change the breakpoints in any procedure in the program simply by using the **procedure** command to change the current procedure to the one whose breakpoints you want to alter. If the procedure is not an active one you will be warned to that effect, but you can still alter its breakpoints.

4.3.5 Program control

You can control the flow of execution of a DAP program from within **psam**: **step** lets you execute 1 or more FORTRAN-PLUS statements; **next** does the same, except that it treats a procedure call as a single statement; **stepi** steps through APAL instructions.

For **step** or **next** to work, when you compile your FORTRAN-PLUS source your invocation of **dapf** has to include the **-g** flag.

step, **next** or **continue** will let you start (or restart) execution and **quit** will let you exit back to the host command line. **continue** *n* lets you continue execution, bypassing all **pause**s and breakpoints, until a total of (*n* – 1) **pause**s and breakpoints have been passed. The total can consist of all **pause**s or all breakpoints, or any mixture of the two.

–g flag slows down normal program execution

One side effect of the **–g** flag and its option to single-step through FORTRAN-PLUS programs is that normal program execution is slowed down slightly; AMT recommends that you recompile your FORTRAN-PLUS program without the **–g** option when you have finished program development.

4.3.6 Access to source code

You can **list** the whole of the current file of FORTRAN-PLUS source statements from within **psam**, as well as display the current line, the line at which execution is suspended, or a range of lines.

In the example program, if you are in procedure **add3** – where we were when we added another breakpoint – issuing a **list** would give you the display:

```
psam: list
File ./fadd3.df
  1    :     function add3(im)
  2    :
  3    :     integer add3(*size(im,1),*size(im,2)),im(*,*)
  4 b1 :     add3=(im+3)
  5>   :     pause 1
  6 B2 :     return
  7    :     end
 psam:
```

Figure 4.9 **psam** *display from the* **list** *command*

Here **B2** shows the breakpoint that we added earlier, and **b1** shows the breakpoint that was added, then disabled. If we enabled the first breakpoint, it would be shown in a **list** as **B1**. The **>** shows where execution has halted, after the **step 3** comand that we issued earlier.

You can list only part of a file by specifying start or end points to **list**. For example **list 3,** would list from line 3 to the end, **list 3,5** would list lines 3, 4 and 5, **list ,.** would list from start of file (line 1) to the current list line, and **list .** would list only the current list line.

Caution

Note that **list** has its own idea of the current line. The first time you use **list** once execution has halted, the current list line is the current line. After that, the current list line is the last line **list** listed. If you explore the active procedures in a program with **top**, **up**, **down** or **procedure** (see later), the initial current list line is the most recently executed line in the procedure. If you change files with **file**, then initially the current list line is the first line in the file.

file lets you change the current source file, letting you **list** included and other source files. In fact **file** will let you change the current file to any file. Hence, you can **list** the contents of other kinds of file, such as macro files, files containing details of breakpoints, and so on.

If you use **procedure**, **top**, **up** or **down** (discussed later), the current file will change automatically to the one holding your newly-selected procedure, and the current list line will change too. However, the current line will not change, and will stay at the line at which execution will start again if you **step**, **next** or **continue**.

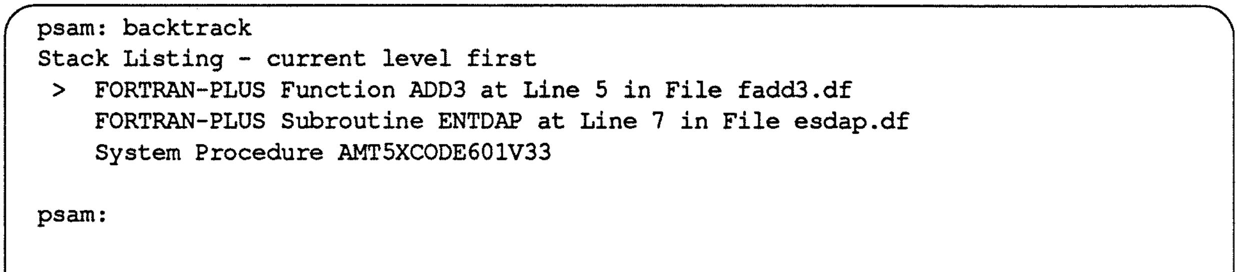
4.3.7 Machine-level commands

stepi lets you step through APAL instructions. **code** lets you disassemble and display APAL code, giving an APAL equivalent to what **list** offers for FORTRAN-PLUS code. **registers** lets you inspect MCU, edge or PE registers, carry and overflow flags in APAL, and the hardware DO loop iteration number. **array** lets you examine data in the array store. If you are debugging APAL code, when control passes to **psam** no current file is selected, although you can use **file** to select and **list** a file.

4.3.8 Stack examination

Once control has passed to **psam**, you can examine the stack in detail, to display FORTRAN-PLUS variables belonging to the different active procedures. **top**, **up**, and **down** let you change the current procedure (FORTRAN-PLUS or APAL) to a different procedure on the stack (that is, to a different *active* procedure). As mentioned above, **procedure** lets you change the current procedure to a non-active procedure as well as to an active procedure. **backtrack** displays details of all the procedures on the stack.

In the example program where execution had stopped at line 5 in **add3**, if you issued a **backtrack** you would get the display:

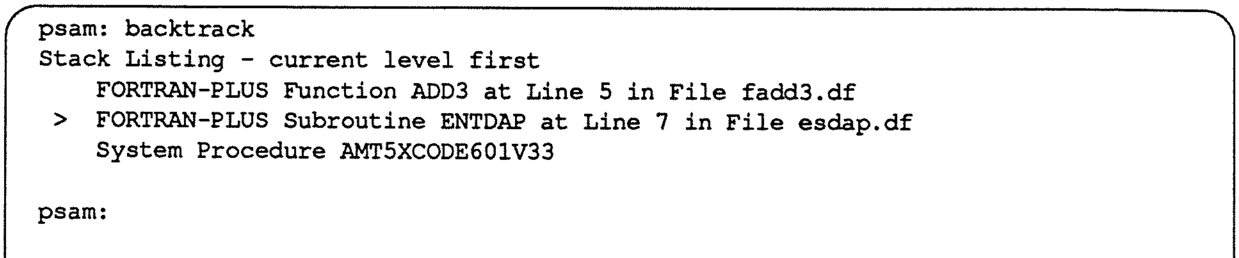


```
psam: backtrack
Stack Listing - current level first
 >  FORTRAN-PLUS Function ADD3 at Line 5 in File fadd3.df
    FORTRAN-PLUS Subroutine ENTDAP at Line 7 in File esdap.df
    System Procedure AMT5XCODE601V33

psam:
```

Figure 4.10 **psam** *display after a* **backtrack** *command*

If the current procedure is active, it is shown on the backtrack, and its entry is flagged by a **>** .

If you then issued a **down**, the current procedure would change to **entdap**, and the **backtrack** display would change to:



```
psam: backtrack
Stack Listing - current level first
    FORTRAN-PLUS Function ADD3 at Line 5 in File fadd3.df
 >  FORTRAN-PLUS Subroutine ENTDAP at Line 7 in File esdap.df
    System Procedure AMT5XCODE601V33

psam:
```

Figure 4.11 **psam** *display after another* **backtrack** *command*

As mentioned above if you use **procedure**, **up**, **down** or **top**, in a FORTRAN-PLUS program the current file and the current list line will change (but not the current line).

Just as **file** is a **psam** command that lets you change the file that you can examine in **psam**, but has no effect on program execution, so **psam**'s **procedure** and **backtrack**, **top**, **up** and **down** have no effect on program execution if and when you restart your DAP program.

4.3.9 Environment variables

You can use *environment variables* to control certain aspects of **psam**'s and **dapdb**'s operation. Note that these variables are *not* the same as the UNIX environment variables, such as **DAPSIZE** or **DAPCP8**.

You change the values of the **psam** environment variables using the **set** and **unset** commands. The **save** command saves the current values of the variables to the file **.defaults** in your home directory. These values then become the defaults each time you start a debugging session – or until you change and save them again. You can also change the defaults by editing the **.defaults** file, using the SunView® tool DefaultsEdit®[1].

environment variables

Details of the environment variables are:

Alias_file

- **Alias_file** gives the name of a file (or names of files, separated from each other by at least one space) holding aliases for **psam** commands. These aliases take effect at the start of each **psam** session.

More

- **More** set to **true** specifies that output to screen is to be displayed a screen at a time, using the UNIX **more** filter. With **More** set to **true**, if you press the space bar the display will scroll up a screenful. If you press <RETURN>, the display will scroll up 1 line, and if you press <Q> you will return to the **psam** prompt.

[1] SunView and DefaultsEdit are registered trademarks of Sun Microsystems Inc

Nearly all the other features of the UNIX **more** are available at the **More** prompt; use <H> to see **More**'s help screen for details.

Order

- **Order** is a list of integers which specifies the order in which the dimensions of an array should be printed, with the first in the list cycled the fastest.

  The default for **Order** depends on the mode of the array being printed:

  - A FORTRAN-PLUS matrix variable is printed with the second dimension (the column) cycling fastest, followed by the first, and if there are any other dimensions, the third, the fourth, and so on until all dimensions are printed. For FORTRAN-PLUS matrix variables **Order = (2 1)** gives the same effect as the default.

    The result is to print the first row of the matrix, followed by the second row, and so on until the whole matrix is printed. For matrix arrays, subsequent matrices are printed in the same way; the printing order of the dimensions being as discussed in the paragraph immediately above.

  - All other arrays are printed with the first dimension cycling fastest, followed by the second, then if there are any other dimensions, the third, the fourth, and so on until the whole array is printed. For non matrix arrays **Order = (1)** gives the same effect as the default.

Pattern_mode

- **Pattern_mode** set to 1 or 2 specifies that logical and characters arrays are to be displayed as 1-dimensional or 2 dimensional grids respectively, instead of like terms being collected.

Source_path

- **Source_path** specifies the path name (or path names, separated from each other by at least one space) of the directories to be searched for macro files, **Alias_files** (see above) and for files specified in the **file** command (see section 4.3.6 on page 67).

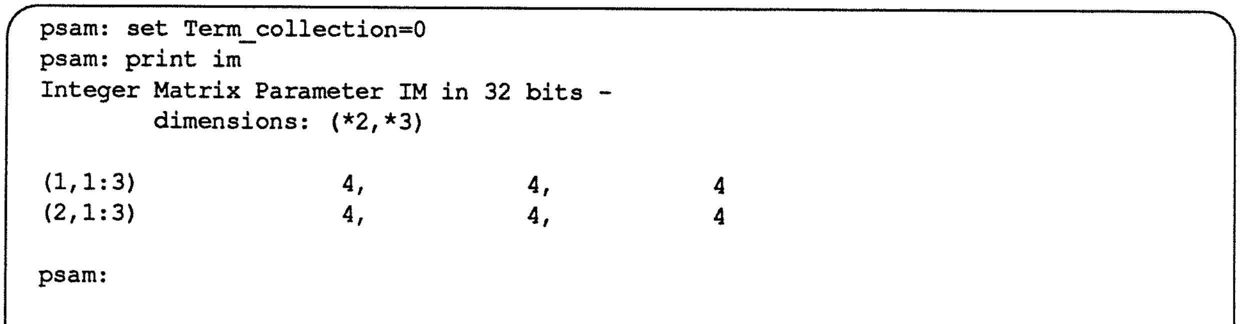
Term_collection

- **Term_collection** specifies the number of dimensions of a variable in which like terms are to be collected when you **print** the variable.

  For the screen dump in figure 4.4 on page 64 **Term_collection was** set to its default, but you could change it to show all the components of variable **im** separately, by using:

  ```
  set term_collection=0
  ```

  Once you have done that, if you move the current procedure back to **add3** with an **up**, you can then see the difference, by issuing a print **im**, which would give the display:



```
psam: set Term_collection=0
psam: print im
Integer Matrix Parameter IM in 32 bits -
        dimensions: (*2,*3)

(1,1:3)             4,            4,            4
(2,1:3)             4,            4,            4

psam:
```

Figure 4.12 **`psam`** *display from the* **`list`** *command*

`Window_width`

- **`Window_width`** specifies the width in characters of the display of the contents of variables.

4.3.10 Miscellaneous commands

You can put a list of **`psam`** commands in a file, and execute them using the **`macro`** command; **`echo`** and **`date`** can be used to echo arguments and the date to the screen (useful in macro files); you can access comprehensive **`help`** facilities; you can re-display the initial diagnostic **`message`** on entry to **`psam`**.

4.3.11 Command line interpreter

`psam` applies a consistent command line interpretation to commands typed in at the **`psam`** prompt and to commands in a macro file – with one exception: the history commands (see below) are not available in macros.

The command line facilities available are:

> and >> to redirect screen output to a file

Many **`psam`** and **`dapdb`** commands generate output, which is normally displayed on the host screen. **`psam`** and **`dapdb`** have a UNIX-like redirection-of-output facility, letting you re-direct the output to a file (**>**), or append it to the end of a file (**>>**).

When you use the construct:

command **>** *file-name*

the output of command *command* is redirected to the file *file-name*. If the destination file already exists its contents will be overwritten by the **`psam`** output.

command truncation

You can truncate any **`psam`** command, so long as the shortened form is unambiguous.

;

You can have several **`psam`** commands on one line, provided they are separated by semi-colons.

#

Any line starting with a **#** is treated as a comment. If **`psam`** finds a **#** in the middle of a line, it treats it as a hexadecimal prefix for a number which it expects to follow the **#** .

aliases

A UNIX-like alias facility is available in **`psam`**; you can create **`alias`**es for use in the current session, and can save them to an **`Alias_file`** for future sessions. You can also **`unalias`** existing aliases.

history commands

Some C-shell-like history commands can be used in **psam**, although the commands cannot be used in a macro. **history** displays a numbered list of the commands used so far in the current session.

quoted commands

You can issue a **psam** command within a pair of "s, to stop unwanted re-direction or history substitution.

For example, suppose you want to insert a breakpoint at line number 5 in the current file, and save the value of variable **im** to file **resultfile**. If you type the command:

```
stop at 5  print im > resultfile
```

then the redirection operator **>** would take precedence, the command line interpreter would redirect the output of **stop at 5 print im** to **resultfile**, and the screen message you normally get when you issue a **stop** command would be sent to file **resultfile** instead. To get the result you want you need to surround the **psam** command to be executed after the breakpoint is reached with " ".

The command:

```
stop at 5 "print im > resultfile"
```

will give you what you want.

4.4 Analysing dump files (dapdb)

All the commands discussed briefly in section 4.3 above are available in both **psam** and **dapdb**, except that those concerned with program flow are only available in **psam**. Commands affected are those that set and use **breakpoints** and **step** and **continue**; **code**, the APAL disassembly command, is only available in **psam**.

difference between psam and dapdb

The essential difference between **psam** and **dapdb** is that **psam** is an on-line debugger, while **dapdb** is for analysing dump files at a later date. **dapdb** is entirely confined to the host, and examines a dump taken of the DAP state, the dump being held in host filestore. **dapdb** does not use any DAP resources. **psam**, however, although it also runs on the host, *does* use DAP resources, in that your DAP program's allocation of array and code store are not released when program execution halts and control is passed to **psam**.

take a dump with dump or <CONTROL-\>

You can take dumps for later **dapdb** analysis either with the **dump** command from within **psam**, or by typing <CONTROL-\> while a DAP program is running. If you do type <CONTROL-\>, then you will normally also get a core dump of your host program.

In addition, dumps are created automatically when a run-time error occurs and the **dapopt -e** option is set to **d** or **dc**. See section 3.4.3 on page 45 for more information on **dapopt** options.

Dump files created when you type <CONTROL-\> are named **dapcore**. Dump files created automatically or from within

psam are named *dof-file-name***.dr**. Here *dof-file-name* is the name of your DOF file that was executing when the dump was taken.

The first dump in a **psam** session will overwrite the contents of the target dump file if it exists already. Second and subsequent dumps in the same session are, however, appended to the same file.

4.4.1 Entering dapdb

You enter **dapdb** by typing at the host prompt:

host% **dapdb** *core-file-name*

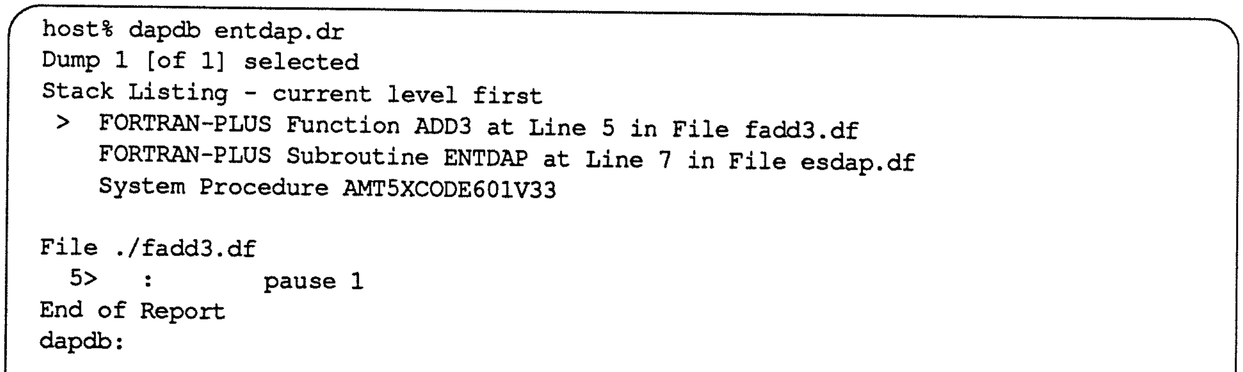
where *core-file-name* is the name of a file that contains one or more dumps of the DAP state. If you do not provide a *core-file-name*, **dapdb** will prompt you for one. If you do not supply a dump file name, **dapdb** will continue to prompt you for it; to escape back to the host command line, type <CONTROL-C>.

Once it has a dump file name **dapdb** loads the file, selects the most recently dumped DAP state from the file, and tells you how many DAP dumps are in the file.

Suppose you took a **dump** of the example program from within **psam**, using the command **dump**. To explore the dump, you would first **quit** from **psam**, then enter **dapdb** with a command (in this case) of:

dapdb entdap.dr

Your screen display might then look like the display:



```
host% dapdb entdap.dr
Dump 1 [of 1] selected
Stack Listing - current level first
 >  FORTRAN-PLUS Function ADD3 at Line 5 in File fadd3.df
    FORTRAN-PLUS Subroutine ENTDAP at Line 7 in File esdap.df
    System Procedure AMT5XCODE601V33

File ./fadd3.df
  5>   :       pause 1
End of Report
dapdb:
```

Figure 4.13 Initial display on entering **dapdb**

You could then explore the dump with **print**, **array**, **backtrack**, and so on – much as you could have explored the DAP state from within **psam**.

4.4.2 dapdb-only commands

There are two commands that are only available from within **dapdb**. **core** lets you select a dump file for examination, **select** lets you select from several DAP dumps that might be in the current dump file.

Section 4.7 , starting on page 80, has the details of all the **dapdb** and **psam** commands.

4.5 trace

4.5.1 Introduction

Both APAL and FORTRAN-PLUS have language-defined **trace** facilities, which output at run time the values of specified data items. AMT does not recommend that you use **trace** as a standard output facility, since it involves the time-consuming overhead of returning control to the host until the output is complete. When output from a **trace** is complete, execution of the DAP program starts again, at the statement or instruction immediately after the **trace**.

Not all **trace** statements in a DAP program are necessarily executed; there are two stages of 'filtering out' – at compile or assembly time, and at run time. All **trace** statements include a trace level number. Options to **dapf** (the FORTRAN-PLUS compiler) and **dapa** (the APAL assembler), and to **dapopt** (the run-time options program) interact with the trace level number to produce **trace** output only when requested.

4.5.2 FORTRAN-PLUS trace

The FORTRAN-PLUS **trace** statement is of the form:

trace *trace-level-number* (*variable-name$_1$, ... variable-name$_i$, ... variable-name$_n$*)

where *trace-level-number* is an integer in the range 1 to 5, and specifies the level of the **trace** statement; *variable-name$_i$* is the variable whose value is to be output when the **trace** statement is executed (see chapter 15 of *DAP Series: FORTRAN-PLUS enhanced*, [2], for more details).

effect of the -D option to dapf

The value of the **-D** option to **dapf** is important if you have any **trace** statements in your source. If you do not specify **-D** when you run **dapf**, or if you specify **-D2**, then **trace** statements are compiled and executed, subject to the restrictions discussed below. If you specify a **-D0** or **-D1** option to **dapf**, your DAP program is still compiled. However, if the **-t** option is also used, with a value greater than 0, a warning message is displayed telling you that the **-D2** option is necessary if **trace** is to be used, and that **-D2** was assumed for the compilation. (This change to full diagnostics will apply to all procedures compiled in that invocation of **dapf**.)

effect of the -t option to dapf

During compilation, only those **trace** statements with a *trace-level-number* less than or equal to the value of the **-t** option to **dapf** will be compiled in. The **-t** default is 0, meaning that by default no FORTRAN-PLUS **trace** statements are compiled.

effect of the -f option to dapopt

Once your DAP program has been compiled, before you run the program, you can specify a run-time **trace** level in a call to **dapopt**. Any compiled **trace** statements with a *trace-level-number* less than or equal to the value of the **`-f`** option to **`dapopt`** will generate a display. The **`-f`** default is 5, meaning that by default all *compiled* FORTRAN-PLUS **`trace`** statements are executed.

The form of the FORTRAN-PLUS **`trace`** diagnostic report is discussed in section 4.6.1 below.

4.5.3 APAL `trace`

The APAL **`trace`** instruction is of the form:

`trace` *trace-number* [*registers-trace-item*] **`level`** *trace-level-number* [*array-store-trace-item*]

where *trace-level-number* is an integer in the range 1 to 15, and specifies the level of **`trace`**ing to be assembled in to the DAP program. For further details, see section 8.1.2 of *DAP Series: APAL Language*, [3].

effect of the `-t` option to `dapa`

During assembly, only those **`trace`** instructions with a *trace-level-number* less than or equal to the value of the **`-t`** option to **`dapa`** will be assembled. The **`-t`** default is 0, meaning that by default no APAL **`trace`** statements are assembled.

effect of the `-a` option to `dapopt`

Once your DAP program has been assembled, before you run the program, you can specify a run-time **`trace`** level in a call to **`dapopt`**. Any assembled **`trace`** instructions with a *trace-level-number* less than or equal to the value of the **`-a`** option to **`dapopt`** will generate a display. The default is 15, meaning that by default all *assembled* APAL **`trace`** statements are executed.

The form of the APAL **`trace`** diagnostic report is discussed in section 4.6.2 below.

4.6 Diagnostic reports

Diagnostic reports are displayed on the host screen (or re-directed to a file) when, during program execution:

- A run-time error occurs
- A **`stop`** or 'active' **`pause`** statement is executed.
- A **`trace`** statement or instruction is executed
- An 'active' **`psam`** breakpoint is reached
- The target of a **`next`**, **`step`** or **`stepi`** command is reached

Note that if you have issued a **`continue`** *n* command (see page 84 for details of **`continue`**), then execution does not stop until the total number of **`pause`** statements and breakpoints encountered since **`continue`** was issued exceeds *n* – or a run-time error is encountered, or control reaches the end of the program first! All **`pause`** statements and breakpoints are 'active' unless they are skipped over by a **`continue`** *n*.

Many **psam** commands also generate diagnostic information, as do commands in **dapdb**.

4.6.1 Reports from FORTRAN-PLUS code

In FORTRAN-PLUS programs, for most types of report, the level of detail in a particular report depends on what level of detail was specified for the **-D** and **-d** options when **dapf** and **dapopt** were run (see section 2.7 on page 32 and section 3.4.7 on page 52, respectively for more details).

For example, if you **step**ped the example program **entdap** used earlier in the chapter, you should get a display like:

```
psam: step
User-Defined Pause: Number 1
FORTRAN-PLUS Subroutine ADD3 at Line 5 in File fadd3.df

Stack Listing - current level first
 >  FORTRAN-PLUS Function ADD3 at Line 5 in File fadd3.df
    FORTRAN-PLUS Subroutine ENTDAP at Line 7 in File esdap.df
    System Procedure AMT5XCODE601V33

File ./fadd3.df
  5>   :       pause 1
End of Report
psam:
```

Figure 4.14 Typical FORTRAN-PLUS diagnostic report

In general, most FORTRAN-PLUS diagnostic reports will contain one or more of the following items:

event-details
`FORTRAN-PLUS` *proc-type proc-name* `at Line` *line-number* `in File` *source-file-name*
stack-backtrack
errors
line or *procedure variables*
display-values
source-line
`End of Report`

where:

- *event-details* gives details of the event that caused the output of the diagnostic report. A typical entry might be:

`Run-Time Error:` *error-details*

or

`User-Defined Pause: Number` *n*

or

`Breakpoint` *n*

- *proc-type* is **Subroutine** or **Function** .
- *proc-name* is the name of the current procedure.

- *line-number* is the line number (in the FORTRAN-PLUS source file) on which execution has stopped (line details are only displayed if the **-D** option to **dapf** that was in force was 1 or 2; default is 2).
- *source-file-name* is the filename of the current FORTRAN-PLUS source file (file details are only displayed if the **-D** option to **dapf** that was in force was 1 or 2 – the default is 2 – and **dapf** was run under DAP basic software release 3.2 or later).
- *stack-backtrack* starts with:

```
Stack Listing - current level first
```

and is a list of all the active procedures.

- *errors* displays, if a computational error has occurred, the positions of the components causing the last unsuppressed FORTRAN-PLUS computational error (see **errors**, page 88).
- *line* or *procedure variables* displays the values of variables if a computational error occurs and if the parameter to the **-d** option to **dapopt** that was in force was not 0 (the default is 0). The variables displayed are those on the failing line (**-d1**) or in the failing subprogram as well (**-d2**).
- *display-values* is a list of the values of all the variables requested by the **psam** command **display** .
- *source-code-line* is the line of source code on which execution has halted.

stop statement

Reports output by the **stop** command are of the form:

```
User-defined Stop: Number number at Location offset
```

where:

- *number* is the number associated with the **stop** in your FORTRAN-PLUS source code.
- *offset* is the instruction offset of the **stop** instruction in hexadecimal (in DAP words, of 32 bits) from the start of the object code version of your program.

  This feature is not of much interest to FORTRAN-PLUS programmers, but is valuable for APAL programmers.

If a **STOP** statement is executed, program execution stops, and control returns to the host. DAP and host programs are abandoned and **psam** is not invoked. See *DAP Series: FORTRAN-PLUS enhanced*, [2], for more details of **stop**.

FORTRAN-PLUS `trace` statement

The conditions under which a diagnostic report is produced for a FORTRAN-PLUS **`trace`** statement are discussed in section 4.5.2 above. The report is of the form:

```
FORTRAN-PLUS Trace
FORTRAN-PLUS proc-type proc-name at Line line-number in File source-file-name
values-of-variables-requested-in-trace-statement
End of Report
```

where:

- *proc-type* is the procedure type, either **`Subroutine`** or **`Function`**.
- *proc-name* is the name of the procedure containing the **`trace`** statement.
- *line-number* is the line number in the FORTRAN-PLUS source file of the **`trace`** statement.
- *values-of-variables-requested-in-trace-statement* are the values of the variables specified in the **`trace`** statement.
- *source-file-name* is the name of the file containing the **`trace`** statements (no file details are available if **`dapf`** was run under DAP basic software release 3.1 or earlier).

All components of specified vectors, matrices and arrays are displayed; you cannot **`trace`** any subset of a matrix, vector or array. The FORTRAN-PLUS storage mode appropriate to each variable is assumed by the system, and spurious values will be printed for any variable which is held in incorrect storage mode.

4.6.2 Reports from APAL code

In the APAL assembler **`dapa`**, there is no comparable option to **`-D`** in **`dapf`**, and all diagnostic reports contain the same level of detail.

APAL diagnostic reports will contain one or more of the following items:

```
event-details
Procedure proc-name + offset-value
stack-backtrack
instructions
End of Report
```

where:

- *event-details* gives details of the event that caused the output of the diagnostic report. A typical entry might be:

```
Run-Time Error: error-details
                                                          or
User-Defined Pause: Number n                              or

Breakpoint n.
```

- *proc-name* is the name of the current APAL code section.
- *offset-value* is the offset in hexadecimal (from the start of the code section) of the last instruction to be executed before processing halted.
- *stack-backtrack* starts with:

```
Stack Listing - current level first
```

and is a list of all the active procedures on the stack.

- *instructions* are the last APAL instruction to be executed before processing was halted, and the instruction that will be executed when processing is re-started. If the cause of the halt was a breakpoint, then only the second instruction is displayed.

APAL `trace` statement

The conditions under which a diagnostic report is produced for an APAL **`trace`** statement were discussed in section 4.5.3 above. The report is of the form:

```
APAL Trace
Trace number trace-number
Procedure proc-name + offset-value
contents-of-registers-requested-in-trace-instruction
values-of-array-store-items-requested-in-trace-instruction
End of Report
```

where:

- *trace-number* is the value of the number specified in the **`trace`** instruction in your code.
- *proc-name* is the name of the procedure containing the **`trace`** instruction.
- *offset* is the word offset (in hexadecimal) of the **`trace`** instruction, from the start of the procedure.
- *contents-of-registers-requested-in-trace-instruction* are the contents of the requested MCU, edge and PE registers.
- *values-of-array-store-items-requested-in-trace-instruction* are the contents of the requested array store items.

4.7 Full specification of **psam** and **dapdb** commands

This section is designed for reference, so some information is repeated briefly where appropriate.

The **psam** and **dapdb** on-line **help** facility gives comprehensive details of all commands. The commands are, in alphabetical order:

alias [*alternative-command-name command-line*]

Create the name *alternative-command-name* as an alias for the command(s) given in *command-line*.

A simple form of parameter substitution using **$** is available: you can have one or more **$**s in *command-line* when you define its alias; when you call *alternative-command-name*, if it has any parameters attached, then all the parameters are used to replace each occurence of **$** in *command-line* when it is executed. If no parameters are supplied with *alternative-command-name*, then *command-line* is executed, with the **$**s replaced by null strings. If you supply parameters to *alternative-command-name* when no **$**s are used in *command-line*, then the parameters are added to the end of *command-line* before it is executed. See below for examples.

If no parameters are supplied with **alias**, a list of all the current aliases is displayed. **unalias** lets you delete an alias from the list of current aliases.

saving aliases

If you issue **alias >** *alias-file-name* at the **psam** prompt, all the current aliases will be written to a file. If you set *alias-file-name* as the value of the **psam** environment variable **Alias_file** (see section 4.3.9 on page 69), all the aliases in the file will be instated at the start of subsequent **psam** sessions.

Examples of aliases are:

The command setting up the alias	*An example of the use of the alias*	*... and its 'meaning'*
`alias s step`	`s 10`	`step 10`
`alias fp "file; procedure"`	`fp`	`file` then `procedure`
`alias plane array $ v i8`	`plane 100`	`array 100 v i8`
`alias spec "echo $; list -$, +$"`	`spec 3`	`echo 3` then `list -3, +3`
`alias fred list`	`fred 10,20`	`list 10,20`

Note the second and fourth examples above. If you want to alias multiple commands you will have to enclose these commands in double quotes or the second, and any subsequent, commands will be treated as commands separate from **alias**.

$$\texttt{array}\ \textit{address} \left\{ \begin{matrix} [\,\texttt{w}\,] \\ \texttt{r}\ [\,/\,\textit{start bit}] \\ \texttt{v}\ [\,\textit{rows}]\ [\,\textit{cols}] \end{matrix} \right\} \left[\left[\left\{ \begin{matrix} \texttt{a} \\ \texttt{b} \\ \texttt{c} \\ \texttt{e} \\ \texttt{h} \\ \texttt{i} \end{matrix} \right\} [\textit{size}] \right] [*\textit{count}] \right]$$

Display the contents of the array store; starting at the specified address and assume that the data is stored in **w**ordpack (the default), **r**owpack or **v**ertical format.

Here *address* is $\left\{\begin{array}{c} \textit{name}\ [+\ \textit{plane.row.word}] \\ \textit{plane.row.word}\ [\mathbf{m}\ \textit{n}] \end{array}\right\}$, where *name* is the name of a data section, *plane.row.word* specifies an optional offset or an absolute address, and *n* is in the range 1 to 7, and selects the corresponding register as an optional address modifier.

In vertical format *rows* specifies the range to display, in the form:

firstrow – lastrow

and *cols* specifies the column range to be displayed in the form:

firstcol | *lastcol*

If you omit either *firstrow* or *firstcol*, then the start of the row or column is assumed. If you omit either *lastrow* or *lastcol*, then the end of the row or column is assumed. The default for both is that all rows and all columns are displayed.

You can display the data in **a**ddress format, or as type **b**it, **c**haracter, **r**eal (**e** for exponential), **h**exadecimal (the default) or **i**nteger. If you specify the type, you can also specify the size in bits in the range 1 - 64, with a default of 32. Real and character sizes have to be a multiple of 8 (for reals the minimum is 24 bits). Size, if you specify it, is ignored when the store is diplayed in address format.

You can display either one (the default) or *count* data items starting at the specified item.

examples

array 400.3 prints one 32-bit hex vaue from plane 400, row 3.

array ..4000 i17 *10 prints ten 17-bit integers starting at word 4000.

array mydata+.25 r b12 /5 prints one 12-bit binary value starting at bit 5 in row 25 of **mydata**.

array 0(m5) v e24 27- prints out (*ES*–27) * *ES* 24-bit real values stored vertically from row 27 to the last row, starting at the plane whose address is 0 modifed by the contents of MCU register 5.

array 20 v b1 prints plane 20 as a grid of 1s and 0s – a special case of vertical format know as pattern format.

attributes *variable-expression* [*variable-expression* ...]

Display the attributes of the specified FORTRAN-PLUS variable(s) in the current procedure whose name matches *variable-expression.*

Note that information on local variables is only available for those variables in active procedures.

variable-expression can include wild cards:

*	Matches zero or more alphanumeric characters
?	Matches one alphanumeric character
[*string*]	Matches any one character from the alphanumeric *string*
[c_1–c_2]	Matches any one ASCII character that lies in the range c_1-c_2 inclusive, where c_1 and c_2 are actual characters, not ASCII values.

Examples of possible **`attributes`** commands are:

`attributes *` Give the attributes of all variables in the current procedure.

`attributes [ABC]?` Give the attributes of all variables in the current procedure whose names are two characters long and which start with **`A`**, **`B`** or **`C`**.

`attributes [a-g]?*` Give the attributes of all variables in the current procedure whose names are two or more characters long and which start with any letter in the range **`A`** to **`G`**.

Note that there is no case signifcance in FORTRAN-PLUS variable names; they are mapped to upper case. If you type in variable names in lower case, they are converted to upper case before any ASCII comparisons are made.

`attributes V[1-f]` Give the attributes of all variables in the current procedure whose names are two characters long, start with **`V`**, and whose second character is in the ASCII sequence from **`1`** to **`F`**.

`backtrack` Display a list of the procedures on the stack.

The entry in the list for each procedure gives you the name of the procedure, the line number or instruction offset at which execution was halted or control passed to another procedure, and for FORTRAN-PLUS procedures the name of the file in which the procedure is held. The current procedure is marked with a **>** beside its entry in the display.

`breakpoints` (**`psam`** only)

Display details of all breakpoints which have been set (with the **`stop`** and **`stopi`** commands) but not yet deleted (with the **`clear`** command).

All breakpoints have a unique reference number, which is displayed, along with the procedure, offset and activity (**`enabled`** or **`disabled`**) of the breakpoints. In addition, for FORTRAN-PLUS breakpoints, the file name and line number of the breakpoint locations are displayed.

`clear` $\left\{\begin{array}{c}\textit{breakpoint-reference-number}\\ *\end{array}\right\}$ (**`psam`** only)

Delete the specified breakpoint, or all breakpoints.

WARNING The reference number for a cleared breakpoint is available for re-use by the system.

$$\textbf{code} \left[\begin{array}{c} \textit{instruction-offset} \\ \left\{\begin{array}{c}-\\+\end{array}\right\} \textit{number-of-instructions} \\ . \end{array} \right] \left[\;,\; \left[\begin{array}{c} \textit{instruction-offset} \\ \left\{\begin{array}{c}-\\+\end{array}\right\} \textit{number-of-instructions} \\ . \end{array} \right] \right]$$

Disassemble and display some or all of the current procedure.

You can only disassemble user-written APAL procedures. The default is that disassembly starts at offset 0 and continues to the end of the procedure. Alternatively, you can specify the extent of the disassembly by giving as an argument to **code** the instruction offsets (either upper offset or lower offset or both). These offsets can be relative either to the current instruction (*number-of-instructions* above) or to the start of the code section (*instruction-offset* above). The current instruction is represented using a '**.**'.

You can use hexadecimal offsets or numbers of instructions, but they have to be preceded by **#**, **0X** or **0x**.

Examples of possible **code** commands are:

code Disassemble and display the whole of the current code section.

code 3 Disassemble and display the instruction at offset 3 (decimal).

code 3, Disassemble and display from the instruction at offset 3 (decimal) to the end of the current code section.

code 3, 10 Disassemble and display from instruction at offset 3 (decimal) to the instruction at offset 10(decimal).

code , . Disassemble and display from the start of the current code section to the current instruction.

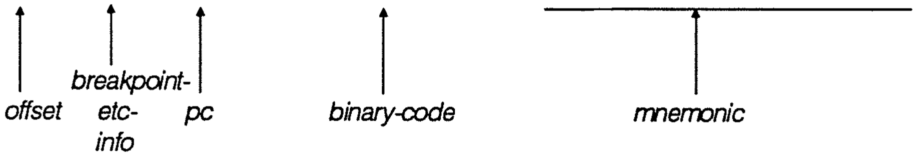
code -#10,+#20 Disassemble and display from the instruction that is 10 (hexadecimal) instructions before, to the instruction that is 20 (hexadecimal) instructions after, the current instruction

When control passes to **psam**, the current instruction is the one that would be executed if and when program execution were restarted. After that, whenever you issue a **procedure**, **up**, **down** or **top** command, the current instruction changes to the last instruction executed in the new procedure. The current instruction also changes when you use the **code** command, when the current instruction becomes the last one you specified to be disassembled. If disassembly includes the current instruction when **psam** was entered, it is indicated by a **-->** character.

Output from the command **code #6C, #74** might be as shown at the top of the next page.

```
6c            0006c:    fd000304          DO   5 TIMES
6d            0006d:    ce001c01     AQ_QQ     E  P  1
6e            0006e:    2e678000     RX        ME  0.0  (ml) (+)
6f    -->     00006f    39688000     CPQRNO    ME
70            00070:    b0d20900     SIC       0  (M2)

71            00071;    0212000      QS        0  (M2)
72       b2   00072:    ca000c01     QQ        N  P  1
73            00073:    88520000     SQ        0  (M2)
74            00074:    6362001f     XR        M3  0.31 (M2)
```

The format of each dis-assembled line of code is:



where:

- *offset* is the offset of the instruction (in hexadecimal) from the start of the current code section (`6c` in the first line in the display above).
- *breakpoint-etc-info* points to the current instruction (at offset #6F above), and any enabled or disabled breakpoints there are (the only breakpoint above is b2, is at offset #72, and is disabled).
- *pc* is the offset of the instruction (in hexadecimal) from the start of the whole loaded program – the program counter value (`0006c` in the first line in the display above).
- *binary-code* is the binary form of the instruction (`fd000304` in the first line in the display above).
- *mnemonic* is the dis-assembled form of the instruction (`DO 5 TIMES` in the first line in the display above).

`continue` [*n*] (**psam** only)

Continue execution of the DAP program. If an argument *n* is given, continue execution, bypassing all **`pause`**s and breakpoints, until a total of (*n*-1) **`pause`**s and breakpoints have been passed. The total can consist of all **`pause`**s or all breakpoints, or any mixture of the two.

`core` *core-file-name* (**dapdb** only)

Change the file to be examined by **dapdb** from the current one to *core-file-name*.

`date`

Display the current time and date.

Re-direction to a specified file allows you to time-and-date stamp an output file.

disable { *breakpoint-reference-number* / * } (**psam** only)

Disable the specified breakpoint or all breakpoints.

Disabled breakpoints don't halt DAP program execution. However, they still appear in the list of breakpoints produced by **breakpoints** and **status**. Disabled breakpoints also appear in the listings produced by **list** and **code**, where they are shown as '**b***n*', where *n* is the breakpoint reference number. You can re-enable disabled breakpoints using the **enable** command, see page 87.

The example of **code** output on page 84 includes a disabled breakpoint, marked as **b2**.

display *variable-name* [**(** *subscripts* **)**]

Display the contents of the specified FORTRAN-PLUS variable (or, optionally, the contents of the subscript-selected components of the specified variable) in the current procedure every time **psam** is entered. If no parameter is specified, show the list of variables to be displayed.

Every time you call **display** with the name of a variable as argument, the variable you give is added to the list to be displayed. You can clear the whole list (but not part of it) with **undisplay**.

Every time **psam** is entered **display** passes the list of variables to be displayed to the **print** command. The only variables whose contents will be displayed are in the procedure in which execution halted; any other variables in the display list – whether local, static or COMMON variables – will not be printed, but will cause output of an error message.

display provides you with a convenient means to monitor the values of variables when, for example, you **step** through a program.

In subscripted references to multi-dimensional variables you can use one or more subscripts to define the range of data to be displayed, much as you do in FORTRAN-PLUS.

For example, suppose a variable **amat** in your program is declared as:

```
amat(*20,*30,4,5)
```

If, every time your program halts, you want all the components of element **amat(,,2,2)** to be displayed, then you can specify in **psam**:

```
display amat(,,2,2)
```

If your interest was only in **amat(10,1,2,2)** then you could specify:

```
display amat(10,1,2,2,)
```

If you wanted **psam** to output elements **amat(,,2,2)**, **amat(,,3,2)** and **amat(,,4,2)**, then **psam** lets you specify:

```
display amat(,,2:4,2)
```

a form of selection not currently valid in FORTRAN-PLUS.

Similarly, if you were interested in components:

amat(10,1,2,2), **amat(11,1,2,2)** and **amat(12,1,2,2)**;
amat(10,2,2,3), **amat(11,1,2,3)** and **amat(12,1,2,3)**; and
amat(10,1,2,4), **amat(11,1,2,4)** and **amat(12,1,2,4)**,

you could specify:

```
display amat(10:12,1,2,2:4)
```

In general, in referring to sub-items of a variable, you use subscripts to define the range of data of interest, and separate your subscripts with commas.

Each subscript takes one of the following forms:

low-index:high-index

or:

index

where these indices specify that only the items (that is, elements or components) between *low-index* and *high-index* inclusive, or only the item *index* should be displayed for the corresponding dimension of the variable. Note that the default values for *low-index* and *high-index* are the first or last item in the given dimension respectively; thus, `20:` displays all items from 20 upwards whilst `:20` displays all items up to and including 20.

You can replace *variable-name* by a *variable-expression* in which the following wild cards can be used:

*	Matches zero or more alphanumeric characters.
?	Matches one alphanumeric character.
[*string*]	Matches any one character from the alphanumeric string *string*.
[c_1-c_2]	Matches any one character in the ASCII character set that lies in the range c_1-c_2 inclusive, where c_1 and c_2 are actual characters, not ASCII values.

examples

The following **display** commands adds all elements and components of the variables as noted, to the list of variables that are be printed when **psam** is next entered:

display *	Add all variables to the list.
display [ABC]?	Add to the list all variables whose names are two characters long and which start with **A**, **B** or **C**.
display [a-g]?*	Add to the list all variables whose names are two or more characters long and which start with any letter in the range **A** to **G**. Note that there is no case signifcance in FORTRAN-PLUS variable names; they are mapped to upper case. If you type in variable names in lower case, they are converted to upper case before any ASCII comparisons are made.
display V[1-f]	Add to the list all variables whose names are two characters long, start with **V**, and whose second character is in the ASCII sequence from **1** to **F**.

You could specify:

```
display a?* (1:3,3,3,3)
```

which would specify that you wanted to have displayed all variables in the then current procedure whose names were at least 2 characters long and started with an `a`, and that you only wanted components `(1,3,3,3)`, `(2,3,3,3)` and `(3,3,3,3)` from each selected variable.

If the procedure included variables declared as:

```
amat(*20,*30,4,5)
avector(*50,3,3,3)
as(4,3,6,4)
asvec(*10,4,4)
apple(*10,*10,2,2,2)
```

then when **psam** was next entered you would see the 3 components you had specified in **display**, but only from the first 3 variables: **amat**, **avec** and **as**. Since the dimensions of **asvec** and **apple** do not match those in the **display** statement, you would get 2 **Invalid subscripts** error messages instead of any values from **asvec** and **apple** .

down

Move down the stack by one procedure, if possible.

If the current procedure is already at the bottom of the stack, or not on the stack, **down** outputs an error message, but otherwise has no effect.

dump (**psam** only)

Copy the DAP state (the whole of the array store part of your DAP program block) to file *dof-file-name*.**dr**, appending to any existing dump(s) created in the current **psam** session. *dof-file-name* is the name of the file containing the DAP object format code being debugged by **psam** . If *dof-file-name*.**dr** is not empty when the **psam** session starts, its contents will be deleted.

echo [*argument* ...]

Display the list of specified arguments terminated by a line feed. (As usual with all commands that produce a display on the host screen, you can re-direct output to a specified file.)

enable $\left\{\begin{array}{c}\textit{breakpoint-reference-number}\\ *\end{array}\right\}$ (**psam** only)

Enable the specified breakpoint, or all breakpoints. Newly created breakpoints are enabled automatically.

Enabled breakpoints halt DAP program execution when control reaches them, and then return control to **psam**; any **psam** commands associated with the breakpoints are then executed.

Enabled breakpoints appear in the listings produced by **list** and **code**, where they are shown as 'B*n*', where *n* is the breakpoint reference number. The examples of **code** output on page 84 and of **list** on page 67 include enabled and disabled breakpoints, marked as 'B*m*' and 'b*n* ' respectively.

errors Display of error information of two kinds:

- *User-recorded errors* – give the contents of any user-defined error recording variables. Use the same display format as would be used by **print**.

  Note that **print** would be controlled by the current values of the **psam** environment variables relevant to a display of logical data of the same mode(s) as the error-recording variables.

  Locations in which errors have occurred correspond to **T** values. If you have not nominated any user-defined variables you will see the display:

```
No user-defined error recording variables
```

- *Any errors* – which will indicate whether there have been any uncleared computational error(s) in the current program.

  This display is of a single character: **T** is reported if there has been an uncleared computational error in the current program, otherwise **F** is displayed. *Any errors* information applies to all computational errors, whether or not you have nominated error recording variables or error interruptiuon masks.

When a computational error occurs in your program, execution stops and **psam** is entered. The initial **psam** display gives you information about the mode and location of that error. If, before you start your program running again, you want to see a repeat of that information, use **message**.

file [*file-name*]

Change the current file to the specified *file-name* ; if no name is specified, print the name of the current file.

The system looks for *file-name* in the list of directories specified by the **psam** environment variable **Search_path**.

help [*topic-name*]

Display help information on the specified command or topic.

Help information is held on all the **psam** and **dapdb** commands; you can see a short introduction to the command line interpreter by typing **help interpreter**. If you don't specify any *topic-name*, a list of all the entries in the **help** database is displayed.

history Display a numbered list of all the commands used so far in the current **psam** or **dapdb** session.

The commands and their effects are:

!!	Repeat the previous command
!*n*	Repeat the *n*th command issued in the current session
!-*n*	Repeat the command issued *n* commands ago

! *str*	Repeat the most recent command which started with *str*
!$	Repeat the last argument used in the previous command
!*	Repeat all arguments used in the prevous command
!^	Repeat the first argument used in the previous command
^*str1*^*str2*	Repeat the previous command, replacing *str1* by *str2*

You can also use the following modifiers with the above commands:

:p	Display but do not execute the command
:s/*str1***/***str2***/**	Replace *str1* in the command by *str2*

examples Hence, if your previous **psam** command had been **help list**, then **!!:s/list/code** would have the same effect as **help code**.

$$\texttt{list}\left[\begin{array}{l}\quad \textit{line-number}\\ \left\{\begin{matrix}-\\+\end{matrix}\right\}\textit{number-of-lines}\\ \qquad .\end{array}\right]\left[\;,\;\left[\begin{array}{l}\quad \textit{line-number}\\ \left\{\begin{matrix}-\\+\end{matrix}\right\}\textit{number-of-lines}\\ \qquad .\end{array}\right]\right]$$

Display part or all of the contents of the current file.

The default is that the entire file is listed. Alternatively, you can specify the extent of the listing by giving either or both of the upper and lower line offsets as an argument to **list**. These offsets can be either relative to the current list line (*number-of-lines* above) or relative to the start of a specified line (*line-number* above). You specify the current list line by using a '**.**'.

You can use hexadecimal line numbers or numbers of lines, but they have to be preceded by **#**,**0X** or **0x**.

examples Examples of possible **list** commands are:

list	List all lines in the file
list 35	List line 35
list 91,100	List from lines 91 to 100 inclusive
list -3,.	List from 3 lines before the current list line, to the current list line
list +3,	List from 3 lines after the current list line to the end of file
list 12,+20	List from line 12 to the line 20 lines after the current list line

list is normally used to list FORTRAN-PLUS source files, but it can be used to list any file selected by **file**. This way you can look at files containing perhaps macros, breakpoints or environment variables.

When control passes to **psam** the current line is the line on which execution was suspended. After that, whenever you issue a **procedure**, **up**, **down** or **top** command, the current line changes to a line in the new procedure. If the new procedure is active, then that line is the one that was being executed when execution was suspended, otherwise the line is line 1. If you use **file** to change the current file, then the new current line is always line 1.

The current line also changes when you use **list**, when it becomes the last line you requested to be displayed.

[**macro**] *macro-file-name*

Execute the **psam** or **dapdb** commands held in the specified file.

Nesting of macros up to 10 levels is possible; if an error occurs while a macro is being interpreted, execution stops and the **psam** or **dapdb** prompt returns. You cannot use history substitution in macros.

If your macro contains commands in which control passes from **psam** or **dapdb** back to your program, once the first such command has been executed, the rest of the macro will be ignored. Such commands include **step**, **continue** or **quit**.

If *macro-file-name* contains breakpoint information generated by the **status** command, then after you have issued **macro** *macro-file-name*, the current file and procedure will be those relevant to the last breakpoint in *macro-file-name*.

$\texttt{map}\begin{bmatrix}\texttt{c}\\\texttt{k}\\\texttt{a}\end{bmatrix}$

Display maps of the occupancy of the MCU and co-processor code stores and the array store of the current DAP program.

Without any parameter the **map** command displays a map of MCU code store occupancy followed by a map of array store occupancy. If the current program includes co-processor code, then an occupancy map of the co-processor code store is included, between the MCU code store and array store maps.

If you supply parameter **a** or **c** then a map of array store occupancy only or MCU code store occupancy only is displayed, respectively.

If you supply parameter **k** and your program includes co-processor code, you will see a map of co-processor store occupancy; if you specify **k**, but your program does not include co-processor code, nothing will be displayed.

Examples of the two types of map are shown below.

MCU and co-processor code store maps

The *MCU code store map* and *co-processor code store map* have the same layout. Each consists of four columns, with one line for each system or user code section. The first column gives the name of the code section. The second and third columns give the start address and size of the code section. Both address and size are in DAP words (32 bits) and given in hexadecimal units. The start addresses are relative to the start of the code store block assigned to the current user program. The fourth column gives the language of user-written code sections.

example

The screen dump at the top of the next page shows you the sort of output you might get if you were debugging the simple FORTRAN-PLUS program **esdap** listed in figure 4.1 on page 61 and you issued a **map c** comand:

```
psam: map c
Code store map:-
Name                                  Start    Size  Language
ENTDAP                                   #0     #6f  FORTRAN-PLUS
ADD3                                    #6f     #59  FORTRAN-PLUS
AMT5XCODE601V33                         #c8     #13  -
AMTVAPSUMMI32CODE                       #db     #b0  -
AMT5SCODE103SV32V01                    #18b    #16d  -
AMT5SADDMS32ICODE                      #2f8     #22  -
AMT5SCODE602V33                        #31a    #13a  -
AMT5SCODE429BV01                       #454     #c4  -
AMT5SADDMM32ICODE                      #518     #23  -

Total Code Store Occupancy : #1000 Words

psam:
```

For FORTRAN-PLUS the fourth column distinguishes between different versions of the FORTRAN-PLUS compiler. Procedures compiled with an earlier FORTRAN-PLUS compiler have their language displayed as `fortran-plus`, otherwise `FORTRAN-PLUS` is displayed to indicate a FORTRAN-PLUS enhanced compiler.

array store map

The *array store map* consists of four columns, with one line for each area. The first column gives the name of the area. The second and third columns give the start address and size of the area. Both address and size are in planes, and the start addresses are relative to the start of the array store block assigned to the current user program. The fourth column gives the access mode of areas corresponding to APAL or FORTRAN-PLUS data sections, or FORTRAN-PLUS literals areas.

example

The screen dump below shows the array store map you would get with a **`map a`** for the same **esdap** FORTRAN-PLUS program:

```
psam: map a
Array store map:-
Name                                 Start    Size  Mode
AMT5AWORK                                0     120  -
AMT5ACONTROL                           120       8  -
AMT5ALITS                              128       1  -
'ROWSCOLS                              129       8  READ ONLY
AMT5PATTERNS                           137       8  READ ONLY
'SYSDAT                                145       2  READ WRITE
'VERSION                               147       1  READ WRITE
'TRACEMESG                             148       8  READ WRITE
AMT5ASTACK                             156     375  -

Total Array Store Occupancy : 768 Planes

psam:
```

masks Display the user-defined FORTRAN-PLUS error interrupt masks.

masks prints the masks in the same way that **print** displays logical data of the corresponding mode – that is, as specified by the environment variable **Pattern_mode** (see **Set** for more details). From your DAP program, you can set locations in your mask(s) to **.FALSE.**, which switches off any error interrupts at locations in variables corresponding to those locations you have marked **.FALSE.** in your mask(s).

Note: Although you can declare a matrix to be, say, **mat(*9000,*6000)**, the DAP processes an *ES^2 sheet* of components of **mat** at a time. If, while the DAP is processing a particular sheet of components, an error occurs that is not suppressed by an error interrupt mask you have nominated, then all processing will stop and control will pass to **psam**. If you have nominated an appropriate error *recording* mask, that mask will record all errors that have occurred in that sheet, but unless you use **psam**'s **contine** command, you cannot be sure that no other errors will occur in the others sheets of **mat** still to be processed. For a fuller discussion of FORTRAN-PLUS's error management facilities, see chapter 15 in [2], *DAP Series: FORTRAN-PLUS enhanced.*

example Suppose you have declared a variable as **EIM(*5,*5)**, and have set the mask to be **.TRUE.** only for components on the leading diagonal, using the FORTRAN-PLUS procedure **PAT_UNIT_DIAG(*n*)**.

You then nominated the variable as an error interrupt mask by calling the FORTRAN-PLUS routine:

```
CALL NOM_EMSK(EIM)
```

If **EIM** is the only error interrupt mask in place, and **PATTERN_MODE** is set to 2 (the default), when you issue the **mask** command you would get the response:

```
psam: masks
User-defined matrix error interrrupt mask -

(1,1:5)    T....
(2,1:5)    .T...
(3,1:5)    ..T..
(4,1:5)    ...T.
(5,1:5)    ....T

psam:
```

message Display the **psam** or **dapdb** entry message first displayed at the start of the current session.

message will make current the file, procedure and line or instruction where execution halted.

next [*number-of-steps*] (**psam** only)

Execute the specified number of FORTRAN-PLUS source statements and then return control to **psam**; if no number is specified, execute one statement. If any procedures are encountered, the whole of each procedure counts as one step.

next only works in procedures compiled with the **-g** option to **dapf**.

If a **next** command is in force, and another interrupt occurs which passes control to **psam**, the effect of the **next** command is cancelled.

Note: **next** *n* is executed as *n* instances of **next**, each instance involving host-DAP communications. As a consequence the time taken to execute a **next** *n* command is greater than the time taken to **continue** between two breakpoints the same *n* statements apart.

print *variable-name* [**(***subscripts***)**]]

Display the contents of the specified FORTRAN-PLUS variable in the current procedure, optionally limiting the display of a multi-element or component variable to one or more elements or components.

In subscripted references to multi-dimensional variables you can use one or more subscripts to define the range of data to be displayed, much as you do in FORTRAN-PLUS.

For example, suppose a variable **amat** in your program is declared as:

```
amat(*20,*30,4,5)
```

If you are in **psam** and want to display all the components of element **amat(,,2,2)**, then you can specify:

```
print amat(,,2,2)
```

If your interest was only in **amat(10,1,2,2)** then you could specify:

```
print amat(10,1,2,2,)
```

If you wanted **psam** to display elements **amat(,,2,2)**, **amat(,,3,2)** and **amat(,,4,2)**, then **psam** lets you specify:

```
print amat(,,2:4,2)
```

a form of indexing not currently valid in FORTRAN-PLUS. Similarly, if you were interested in components:

amat(10,1,2,2), **amat(11,1,2,2)** and **amat(12,1,2,2)**;
amat(10,2,2,3), **amat(11,1,2,3)** and **amat(12,1,2,3)**;
amat(10,1,2,4), **amat(11,1,2,4)** and **amat(12,1,2,4)**

you could specify:

```
print amat(10:12,1,2,2:4)
```

In general, if you want to refer to elements or components of a variable, you use subscripts to define the range of data of interest, and separate your subscripts with commas.

Each subscript takes the following form:

low-index:high-index

or:

index

where these indices specify that only the items (that is, elements or components) between *low-index* and *high-index*, or only the item *index* should be displayed for the corresponding dimension of the variable. Note that the default values for *low-index* and *high-index* are the first or last item in the given dimension respectively. Hence, **20:** displays all items from 20 upwards whilst **:20** displays all items up to and including 20.

You can replace *variable-name* by a *variable-expression* in which the following wild cards can be used:

*	Matches zero or more alphanumeric characters
?	Matches one alphanumeric character
[*string*]	Matches any one character from the alphanumeric *string*
[c_1–c_2]	Matches any one ASCII character that lies in the range c_1-c_2 inclusive, where c_1 and c_2 are actual characters, not ASCII values.

The way in which **print** displays array variables is fixed by the environment variables **Order**, **Pattern_mode**, **Term_collection** and **Window_width**. In logical arrays, components that are **.TRUE.** are displayed as **T**; **.FALSE.** values are displayed as **F** if **Pattern_mode** is 0, and **.** otherwise.

If you specify to be **print**ed any variable that does not exist in the current procedure, or any non-existent element or component of a variable you will get a **No information for these variables** or an **Invalid subscripts** message.

examples Some examples of **print**:

print *	Prints all variables in the current procedure.
print V[0-F]	Prints all elements or components of any variable in the current procedure whose name is **V0**, **V1**, ... , **VE**, **VF**.
print VEC1(7)	Prints the 7th component of vector **VEC1**.
print VEC1 (16:24)	Prints components 16 to 24 of the vector **VEC1**.
print vec1(8:)	Prints from component 8 to the last component of vector **VEC1** .

Note that there is no case signifcance in FORTRAN-PLUS variable names; they are mapped to upper case. If you type in variable names in lower case, they are converted to upper case before any further processing.

If a variable is declared as **mat (*3, *4)** and **psam**'s environment variables have their default values, then **print mat** might produce the display:

```
psam: print mat
Integer Matrix Parameter MAT in 32 bits -
        dimensions: (*3,*4)

(1,1:4)             1,          2,           3,          4
(2,1:4)             5,          6,           7,          8
(3,1:4)             9,         10,          11,         12

psam:
```

`procedure` *procedure-name*

Change the current procedure to the one specified; if no argument is supplied, display the name of the current procedure.

The procedure to be selected does not need to be an active procedure; you can select any procedure in any part of the DAP program and make it current. If you do select a non-active procedure, no local variables can be printed nor their attributes displayed, and a warning message is output.

`quit` Quit **psam** and return control to the host operating system, abandoning host and DAP programs.

$$\texttt{registers}\left[\begin{array}{l}\texttt{a}\\ \texttt{c}\\ \texttt{d}\\ \texttt{f}\\ \texttt{q}\\ \left[\begin{array}{l}\texttt{m}n\\ \texttt{me}\\ \texttt{m*}\end{array}\right]\left[\left\{\begin{array}{l}\texttt{a}\\ \texttt{b}\\ \texttt{c}\\ \texttt{e}\\ \texttt{h}\\ \texttt{i}\end{array}\right\}[\textit{size}]\right]\end{array}\right]$$

Display the contents of some or all of the PE register planes, MCU and edge registers; the carry and overflow flags; and the hardware DO loop iteration number.

You can limit the display to a specified PE register plane (**a**, **c**, **q**), or the carry and overflow flags (**f**), or the hardware DO loop iteration number (**d**), or a specified MCU register (**m***n*), or the edge register (**me**), or all MCU and edge registers (**m***).

If you specify MCU or edge registers explicitly, you can specify the form in which the data is displayed – in the form of an **a**ddress, a **b**it pattern, **c**haracters, a real (e for exponential), a **h**exadecimal or an **i**nteger, with the default of hexadecimal. Optionally, you can specify the size of the displayed data item(s): 24 to 64 bits in steps of 8 for reals, 1 to 64 in steps of 1 for integers or hexadecimals; default size is 32 bits, or *ES* for the edge register. If you do specify size, it has to be less than or equal to the size of the register(s).

You can specify **registers** on its own, in which case you will get a display of all the PE , MCU and edge registers, the carry and overflow flags, and the hardware DO loop iteration number. the format of the regsiters display will be hexadecimal, with a size of 32 bits, and *ES* for the edge register.

Note: A hardware **DO** loop is used in APAL programs, it is not the same as a FORTRAN-PLUS **DO** loop.

`save` Save the current values of the **psam** environment variables to the file **.defaults** in your home directory.

`select` [*dump-number*] (**dapdb** only)

Select the *dump-number*th DAP dump in the current core-file for examination by **dapdb**.

If you don't give an argument, **dapdb** will tell you which dump is already selected.

boolean-name
set *numeric-name* = *value*
list-name = (*list*)

Change the contents of the specified **psam** environment variable. If no arguments are supplied, list the current values of all the environment variables.

The names of the variables, their type, possible and default values are:

Name	*Type*	*Range*	*Default*
Alias_file	list	Any list of file names	**()**
More	boolean	**TRUE**, **FALSE**	**TRUE**
Order	list	The list has to consist of all integers in the range 1 to *n*, in any order, where *n* has any positive value	**default**
Pattern_mode	numeric	**0 – 2**	**2**
Source_path	list	Any list of directory names	**(.)**
Term_collection	numeric	**0 – 7**	**7**
Window_width	numeric	**45 – 132**	**80**

Names of enviroment variables are not case-sensitive, and may be abbreviated as long as they remain unambiguous.

Some examples of the use of **set**:

Command	*Effect*
set Alias_file = (~/*myfile***)**	Installs the **psam** command aliases held in the file *myfile* in your home directory at the start of a subsequent **psam** session. You write the aliases to *myfile* when you issue the **save** command.
set More	Sends screen output through the UNIX filter **more** .
set Order = (2 1)	When arrays of rank 2 or more are being printed, the second dimension will vary fastest, followed by the first, the third, the fourth, and so on, until components from all the dimensions have been printed. The default for **Order** for printing FORTRAN-PLUS matrices is **(2 1)**; for all other arrays the default is **(1 2)**.
set Pattern_mode = 1	Print logical and character arrays as 1-dimensional patterns.
set Source_path = (. ~/*fred***)**	Searches for macro files, **Alias_file** files and current files, first in the current directory, then in the directory *fred* in your home directory.
set Term_collection = 4	When displaying multi-component variables, collect terms only in the first four dimensions being printed.
set Window_width = 60	When displaying variables, use a maximum text width of 60 characters.

status Display the current breakpoints in command format. (**psam** only)

The output from this command is normally redirected to a file, so that you can re-instate the current breakpoints during another **psam** session by executing that file using the

macro command. When in a subsequent **psam** session you re-instate such breakpoints, **psam**'s current file and procedure are those relevant to the last breakpoint you re-instate.

Note that **status >** *filename* records the existence of breakpoints, but not whether they are enabled or disabled. A call in a subsequent **psam** session to **macro** *filename* (or simply to *filename*) will install as enabled all the breakpoints in *filename.*

step [*number-of-steps*] (**psam** only)

Execute the specified number of FORTRAN-PLUS statements and then return control to **psam**; if no number is specified, execute one statement. If any procedures are encountered, each executable statement of each procedure counts as one step.

step only works in procedures compiled with the **-g** option to **dapf**. If a **step** command is in force, and another interrupt occurs which passes control to **psam**, then the effect of the **step** command is cancelled.

Note: **step** *n* is executed as *n* instances of **step**, each instance involving host-DAP communications. As a consequence the time to execute a **step** *n* command is greater than the time to **continue** between two breakpoints the same *n* statements apart.

stepi [*number-of-instructions*] (**psam** only)

Execute the specified number of APAL instructions and then return control to **psam**; if no number of instructions is specified, execute one instruction.

If a **stepi** command is in force, and another interrupt occurs which passes control to **psam**, then the effect of the **stepi** command is cancelled.

Note: **stepi** *n* is executed as *n* instances of **stepi**, each instance involving host-DAP communications. As a consequence the time to execute a **stepi** *n* command is greater than the time to **continue** between two breakpoints the same *n* instructions apart.

stop at *line-number* [*command*] (**psam** only)

Insert a breakpoint in the current FORTRAN-PLUS source file at the specified *line-number.* If *command* is specified, when the breakpoint is reached and control is passed by the run-time diagnostic system to **psam**, execute *command.*

stop in *procedure* [*command*] (**psam** only)

Insert a breakpoint on the first executable line of the specified FORTRAN-PLUS *procedure.* If *command* is specified, when the breakpoint is reached and control is passed by the run-time diagnostic system to **psam**, execute *command.*

stopi at *code-offset* [*command*] (**psam** only)

Insert a breakpoint at the specified *code-offset* in the current APAL procedure. If *command* is specified, when the breakpoint is reached and control is passed by the run-time diagnostic system to **psam**, execute *command.*

stopi in *procedure* [*command*] (**psam** only)

Insert a breakpoint at offset 1 (the normal entry point) in the specified APAL *procedure*. If *command* is specified, when the breakpoint is reached and control is passed by the run-time diagnostic system to **psam**, execute the command.

time Display *execution time* and *execution time difference* in units of machine cycles, for the current program.

Execution time gives the number of cycles used so far by the current program. The value is not incremented when the program is not executing. Execution time includes cycles used by the system in suspending and restarting the program – for example in order to re-enter **psam**. The number of cycles used to suspend and restart is zero on the simulator and has an indeterminate value (about 150) on the hardware.

Execution time difference is the difference in cycles between the value of the current execution time and its value when you last used the time command. If, however, you use time again without having tried to restart your program, execution time and execution time difference are displayed unchanged.

top Make the procedure at the top of the stack the current procedure.

top changes the current procedure, and line or instruction, and, for FORTRAN-PLUS programs, the current file.

unalias [*alternate-command-name*]

Delete *alternate-command-name* from the list of aliases in the current session. If no parameter is supplied, display a list of all the aliases in the current session.

undisplay Remove all variables from the list of variables to be displayed on entering **psam**. An individual variable cannot be **undisplay**ed.

unset [*variable-name*]

Change the value of the specified boolean environment variable to FALSE, or change the value of the specified variable to its system default value. If no variables are specified, display the current values of all the environment variables.

Names of enviroment variables are not case-sensitive, and can be abbreviated as long as they are unique.

Some examples of the use of **unset**:

Command	*Effect*
unset Alias_file	No file of alias commands is actioned at the start of a subsequent **psam** session.
unset More	Screen output is not sent through the UNIX filter **more** .
unset Order	Resets **Order** to **default**, after which FORTRAN-PLUS matrices are printed by cycling the second dimension fastest, then the first, then the

third, then the fourth, and so on. All other arrays are printed by cycling the first dimension fastest, then the second, then the third, and so on.

unset Pattern_mode Logical and character arrays are printed as 2-dimensional grids.

unset Source_path **Macro files, Alias_file** files and current files are searched for in the current directory only.

unset Term_collection When multi-component variables are displayed, terms in all dimensions are collected.

unset Window_width When variables are displayed, uses a maximum text width of 80 characters.

up Move up the stack by one procedure, if possible.

If the current procedure is already at the top of the stack, or not on the stack, **up** outputs an error message, but otherwise has no effect. **up** changes the current procedure, and line or instruction, and for FORTRAN-PLUS programs, the current file.

Chapter 5

CIF file and library maintenance

5.1 Multi-module CIF files

The FORTRAN-PLUS compiler and the APAL assembler produce consolidator input format (CIF) files. There is usually one CIF file produced for each input source file. Each APAL module or FORTRAN-PLUS subroutine or function in the input source file produces a separate CIF module, so a CIF file can contain several CIF modules. A CIF file containing more than one module is called a multi-module CIF file. A concatenation of two or more multi-module CIF files is itself a valid multi-module CIF file.

For simplicity, most of the examples in this chapter assume that the CIF files only contain one module, so the phrase 'module **a.dc**' should really be 'the single module in the CIF file **a.dc**'. Such a CIF file is just a special case of a multi-module CIF file.

consolidator links individual and multi module files

The consolidator will link both individual CIF modules and multi-module CIF files. However in the latter case, the entire contents of the file (which might contain many CIF modules) is linked into the DAP object format (DOF) file even if some modules are not required.

For example, if the multi-module CIF file **all.dc** is a concatenation of individual CIF files **a.dc**, **b.dc** and **c.dc** then the command:

```
    dapf x.dc  all.dc
```

will link the CIF file **x.dc** with **a.dc**, **b.dc** and **c.dc**. If only **b.dc** was actually needed then the resulting DOF file will be unnecessarily large and will not be identical to the file generated by:

```
    dapf  x.dc  b.dc
```

The solution to this inefficient use of DAP memory is to hold CIF modules in a CIF library, and only to link those modules that are needed for a particular program. **daplib**, the CIF library maintenance utility, is used to build and maintain CIF libraries, with the help of index tables it maintains of all the modules in each library.

Hence CIF library files can be used to hold suites of subroutines, without generating redundant code in DOF files.

5.2 The **daplib** command

CIF library maintenance is carried out by the program **daplib**, which you can use to create libraries, add or remove modules and synonyms, and list the contents of libraries. If you want to use the CIF modules on any other than a DAP 500 series machine, you need to set the environment variable **DAPSIZE** to the edge-size of the DAP your program is to run on. Note that the variable is **DAPSIZE** not **dapsize**.

For example:

```
setenv DAPSIZE 64
```

will cause **daplib** to produce a DAP 600 CIF library file. You cannot put CIF modules for different size DAPs in the same CIF library file.

The command:

```
setenv
```

will print the current environment variables.

The command:

```
unsetenv DAPSIZE
```

will clear the **DAPSIZE** variable, leaving the default value of 32 in place.

Caution

If you are using multiple windows on your host, **DAPSIZE** will only affect the windows in which it has been set; for other windows the default value will apply.

5.2.1 Creating a CIF library

You can input to **daplib** individual CIF module files, multi-module CIF files or CIF library files. By default a new CIF library file is created with name **daplib.dl** (**.dl** being the standard extension for CIF libraries). You can send output to a different file by using the **-o** flag.

For example, you could create a CIF library file **dapobj.dl** to hold the 2 modules generated from the simple DAP program used in chapter 4:

```
daplib -o dapobj.dl esdap.dc fadd3.dc
```

that is, **dapobj.dl** would contain the 2 CIF modules **esdap.dc**, and **fadd3.dc**.

You can add further CIF modules to an existing CIF library by specifying the library as one of the input parameters.

Hence:

```
daplib -o dapobj.dl dapobj.dl
                        ftimes2.dc
```

would add **ftimes2.dc** to the modules in the existing library **dapobj.dl**. Note that if you omit **dapobj.dl** as one of the input parameters, the existing library would be overwritten.

That is:

```
daplib -o dapobj.dl ftimes2.dc
```

would result in the library **dapobj.dl** containing only the module **ftimes2.dc**. Note also that if you specify an output filename that does not end with **.dl daplib** will create a file with **.dl** added to the end.

5.2.2 Including and excluding CIF modules

When you use the **daplib** command, not all the modules in an input multi-module CIF file or library need be included in the output library.

-m to include

-f to exclude

You can use the **-m** flag to include only those modules you specify – and you can use the **-f** flag to specify input modules you want to exclude from the output library. You can specify input modules explicitly, or you can use wild cards and regular expressions in the module specification for **-m** and **-f**, and you can refer to a selected module by any of its synonyms.

You can use any of the following wild card characters and regular expressions:

- The wild card ***** matches zero or more characters.
- The wild card **?** matches any single character.
- The regular exprerssion **[***string***]** matches any single character in *string*. For example, **[abc123]** matches **a**, **b**, **c**, **1**, **2** or **3**.
- The regular expression **[***c*$_1$–*c*$_2$**]** matches any single character in the range specified. For example, **[p-s]** matches **p**, **q**, **r** or **s**, and **[2-4]** matches **2**, **3** or **4**.

If you use a wild card to specify a range within a regular expression, **daplib** will issue a warning, and will not match any synonyms. If you use a wild card as one of a string of characters within a regular expression, the wild card is ignored.

For example:	*Expression*	*Match*	*No match*
	'[aeiou]*s'	**AMT5PROGS** **EMPTYSETS**	**DAPSCREENS** **INITDAP**
	'*r_e[a-f]?'	**FREDA** **BTREE1**	**FRED** **NREVN**

As with all **daplib** options, the module matching is applied to each input file or library in turn.

You can match modules against more than one expression in a single **daplib** command by repeated use of the **-m** flag. In this case all the modules matched by each expression are candidates for inclusion in the output library.

For example:

```
host# daplib -m '[a-c]????' -m '*graphics' fred.dc bill.dl
```

matches those modules in **fred.dc** and **bill.dl** which have a synonym which either starts with **A**, **B** or **C** and has 5 characters in all, or ends with the string **GRAPHICS**. Note the use of quotes surrounding wild card expressions, to prevent the UNIX shell trying to match them with filenames. The selected modules are included in the default output library **daplib.dl**.

using -f

You can use the **-f** flag in a similar way, to exclude input modules from the output file. Hence:

```
host# daplib -f 'red*.dc' oldcif.dl -0 newcif.dl bluewing.dc
```

will create an output libray **newcif.dl**, which will include module **bluewing.dc**, and all modules from the library **oldcif.dl** except those whcih have a synonym that starts with **red** and ends with **.dc**.

5.2.3 Synonyms

You can give one or more aliases for the name of a CIF module. Together the module name and its associated aliases form the synonyms of the module. To add a synonym to a module, you use the **-D** flag to **daplib**. For example,

```
    daplib -o dapobj.dl dapobj.dl
                                  -D maths=add3
```

would add the synonym **maths** for the CIF module **add3** in the CIF library **dapobj.dl**.

in FORTRAN-PLUS

In fact, FORTRAN-PLUS users will not normally need to use synonyms in a CIF library. This is because each FORTRAN-PLUS subroutine or function is converted to a single CIF module having the same name as the subroutine or function.

in APAL

An APAL module often has several aliases associated with it. These aliases normally have the same names as the entry points into the module. If the name of an entry point is not also a synoynm of a module in the CIF library, the consolidator will not be able to find the module. So, by adding synonyms to APAL modules held in CIF library format, you can make the different entry points in the module accessible to other APAL code section.

You can remove syonyms by using the **-x** flag, and you can use the same wild card and regular expressions that you can use with **-m** and **-f** – for more details see section 5.2.2 on page 103. The synonym(s) given after the flag, or that match the given expression, do not appear in the output library – if a synonym is the only synonym of a particular module, then that module is deleted from the library – and you are given a warning message on the host screen.

Synonyms have to be unique within a CIF library. If **daplib** encounters a duplicated synonym it will normally treat it as an error and not produce an output file. You can use the **-k** flag to force **daplib** to remove any synonyms from a module you want to add to a library, if that synonym already exists in the library. Once again, you can use the same wild cards and regular expressions that **-m** and **-f** can use to specify the synonyms to be removed. If a synonym is the only one a particular module has, then the module is not included in the output file – and you see a warning message on your host screen.

5.2.4 Listing CIF library contents

The **-L** flag requests **daplib** to display synonyms and module names of either the input files or the output library, or both. The parameter after the **-L** flag has to be an integer in the range 1 to 3, specifying the listing required. The effect of the different values is that:

- **1** Lists the output library only.
- **2** Lists the input files only.
- **3** Lists the input files and output library.

An example of a level 3 listing and the command that generated it is shown on the next page.

```
host%  daplib -o dapobj.dl esdap.dc fadd3.dc -D maths=add3 -L 3

DAP Library Utility 4.0S    (c) Copyright AMT 1987     Fri Nov 16 15:54:52 1990

Maintaining libraries for DAP 500 series

                             **  INPUT FILES  **

Synonym
          Module                              L    V    Creation date

                       CIF File:  esdap.dc
ENTDAP
         ENTDAP                              F   4.0S  Fri Nov 16 15:43:00 1990

                       CIF File:  fadd3.dc
ADD3
         ADD3                                F   4.0S  Fri Nov 16 15:43:00 1990


                             ** OUTPUT LIBRARY **
Synonym
         Module                              L    V    Creation date

ADD3
         ADD3                                F   4.0S  Fri Nov 16 15:43:00 1990
ENTDAP
         ENTDAP                              F   4.0S  Fri Nov 16 15:43:00 1990
MATHS
         ADD3                                F   4.0S  Fri Nov 16 15:43:00 1990


CIF library created : dapobj.dl
```

Every synonym in the multi-module file or library being listed is given together with the name of the corresponding module, the source language (F for FORTRAN-PLUS, A for APAL), the version number of the compiler or assembler which created it and the creation date and time. These listings are sent to the standard output stream – usually your host screen. If there are any diagnostics (comments, warnings or errors), then explanatory messages are included. A one line summary of all diagnostics is also sent to the standard error stream – again usually the host screen.

The **-y** flag suppresses the generation of the output library; you can use it in conjunction with the **-L** flag if you want to list the contents of a library, but not to change it. For example:

```
daplib -y -L2 dapobj.dl
```

would list the synonyms and modules of **dapobj.dl** without changing it.

5.2.5 Interaction of `daplib`'s `-m`, `-f`, `-d` and `-x` options

You can use any or all of the **`-m`**, **`-f`**, **`-D`** and **`-f`** flags any number of times. Regardless of the position of the options on your command line, the **`-m`**, **`-f`**, **`-D`** and **`-x`** options are always applied in the order:

```
-m -f -D -x
```

Some modules you have selected using the **`-m`** flag might not be included in the output library, since they might be filtered out later by the **`-f`** option. Again some modules might lose all their synonyms after the **`-x`** option is applied – although **`daplib`** will warn you that those 'no-synonym' modules have not been included in the output library.

5.3 `daplib` flags

This section contains a summary of all the **`daplib`** flags.

`-D` *syn=name* Define an additional synonym *syn* for an existing module with the synonym *name* .

`-f` *syn-exp* Filter out from the input files any module with a synonym matching *syn-exp*, and do not place it in the output library. The filter is applied to each input file in turn.

syn-exp can include one or more specific synonym names, and can include a combination of wild cards and regular expressions – for details, see section 5.2.2 on page 103.

`-k` Kill (remove) any second or subsequent occurrences of any synonyms in the input files. If a removed synoym is the only synonym of an input module, that module is not included in the output library, and a warning message is sent to standard output.

`-L`*n* Generate a **`daplib`** listing of the level specified by *n*. Valid values of *n* and the effects they have are:

1 Lists the output library only.

2 Lists each input file only.

3 Lists both input files and the output library.

`-m` *syn-exp* Only copy a module from an input file to the output library if the module has a synonym matching *syn-exp*.

syn-exp can include one or more specific synonym names, and can include a combination of wild cards and regular expressions – for details, see section 5.2.2 on page 103.

`-o` *name* Generate an output library called *name*.**`dl`** instead of the default name **`daplib.dl`** . If *name* ends with **`.dl`**, do not add a further **`.dl`** .

`-x` *syn-exp* Delete any synonym matching *syn-exp* from the output library. If *syn-exp* matches the only synonym of an input module, that module is not included in the output library, and a warning message is sent to standard output.

syn-exp can include one or more specific synonym names, and can include a combination of wild cards and regular expressions – for details, see section 5.2.2 on page 103.

`-y` Inhibit the production of an output library.

5.4 Linking with CIF libraries

The main advantage of using CIF libraries instead of multi-module CIF files is that the consolidator will extract only the modules actually required from a library. To do this the consolidator scans the library index to locate missing external references. The order in which the libraries and other CIF files are specified is significant as the consolidator uses a rigid search algorithm when looking for unsatisfied external references (UERs).

UERs

The consolidator maintains a list of UERs and tries to resolve them when each new input file is read, in the following way:

- If the file is a (multi-module) CIF file, all procedures in it are available to resolve references (since all modules in the file are linked in)
- If the file is a CIF library any module in it which satisfies a UER is linked in

Obviously either of the above can introduce new UERS and in this case the current input file is rescanned in an attempt to resolve them. This process is repeated until no new UERs are introduced. The next input file is then read, and the process repeated.

For example, suppose **dapfort.dc** references a module **c.dc** which in turn references a module **f.dc**, which are both in the CIF library **daplib.dl**. That is:

```
dapfort.dc - - - - - -> c.dc - - - - - - -> f.dc
```

Hence, during the search process:

the consolidator starts at the beginning of the index of **daplib.dl** and steps through till it finds module **c.dc**. **c.dc** references **f.dc**, so the consolidator restarts at the beginning of **daplib.dl**'s index and steps through again looking for **f.dc**, and so on.

```
daplib.dl

a.dc
b.dc
c.dc
d.dc
e.dc
f.dc
.
.
```

The consolidator loops round and round in this way trying to satisfy all the UERs. When the CIF library file has been searched for each UER the consolidator moves on.

Once the consolidator has finished scanning a CIF library it will not return to it. This makes the file order very important. For example, if you want to compile and link two files **dapfort1.df** and **dapfort2.df**, each of which references a CIF modules such that:

```
dapfort1.df - - - - > b.dc

dapfort2.df - - - - > c.dc
```

and the modules **b.dc** and **c.dc** are contained in two CIF libraries:

```
daplib1.dl        daplib2.dl
```

which contain modules:

```
b.dc              c.dc
.                 .
.                 .
```

then the command:

```
dapf  dapfort1.df  daplib2.dl  dapfort2.df  daplib1.dl
```

results in a DOF file **d.out**, which contains modules:

```
dapfort1.dc
dapfort2.dc
b.dc
```

Module **c.dc** is missing from the DOF file **d.out**.

An analysis of the actions of the consolidator shows why. The consolidator:

- Links **dapfort1.dc**, having one UER, to module **b.dc** .
- Searches file **daplib2.dl** trying to satisfy the UER to module **b.dc**. Having failed to find **b.dc** the consolidator moves to file **dapfort2.dc** .
- Links **dapfort2.dc**, and now has two UERs **b.dc** and **c.dc** .
- Searches file **daplib1.dl** trying to satisfy both UERs.
- Finds module **b.dc** and links it, fails to find **c.dc** and ends.

This shows that the action of the consolidator is an one-way process; once it has exhaustively searched a CIF library and moved on, it does not re-open the same CIF library again. However, the consolidator does try to satisfy any UERs generated by earlier CIF files, at each stage of the process.

If the above example was replaced by:

```
dapf  dapfort1.df  daplib1.dl  dapfort2.df daplib2.dl
```

then both **b.dc** and **c.dc** would be found.

Alternatively:

```
dapf  dapfort1.df  dapfort2.df  daplib1.dl  daplib2.dl
```

would also be successful.

Note that a CIF library should never be the first filename in a **dapa** or **dapf** command, because at that stage the consolidator has no external references to satisfy. It is important that CIF libraries are named in the correct order, particularly if there are several versions of the same module in different libraries and a specific version is required.

Chapter 6

APAL assembly system

The process of generating a DAP object format (DOF) file, which can be loaded and run on the DAP, from one or more APAL source files is carried out by the APAL assembly system.

6.1 Producing APAL programs for various DAP models

The APAL assembly system can produce object code for DAP 500 or 600 series machines, with or without co-processors. Two environment variables, **DAPSIZE and DAPCP8**, let you specify what DAP model you want to generate object code for. You set environment variables using the command **setenv**, and delete them using **unsetenv**. For example:

```
setenv FRED xyz
```

sets the value of the environment variable **FRED** to **xyz**, and:

```
unsetenv FRED
```

deletes **FRED** from the environment. **setenv** on its own:

```
setenv
```

lists all the current environment variables, and their values.

DAPSIZE lets you specify whether you want to generate code for a DAP 500 or DAP 600 machine, and takes the value **32** or **64** (the DAP edge size).

DAPCP8, if it has the value **yes**, specifies that you want code for a DAP with co-processors.

So the commands:

```
setenv  DAPSIZE 64
setenv  DAPCP8  yes
```

tell the APAL assembly system that you want to generate code for a DAP 600C series machine.

If neither **DAPSIZE** or **DAPCP8** are set to a recognised value – or you have not given them a value – then the default action is to generate code for a DAP 500 without co-processor.

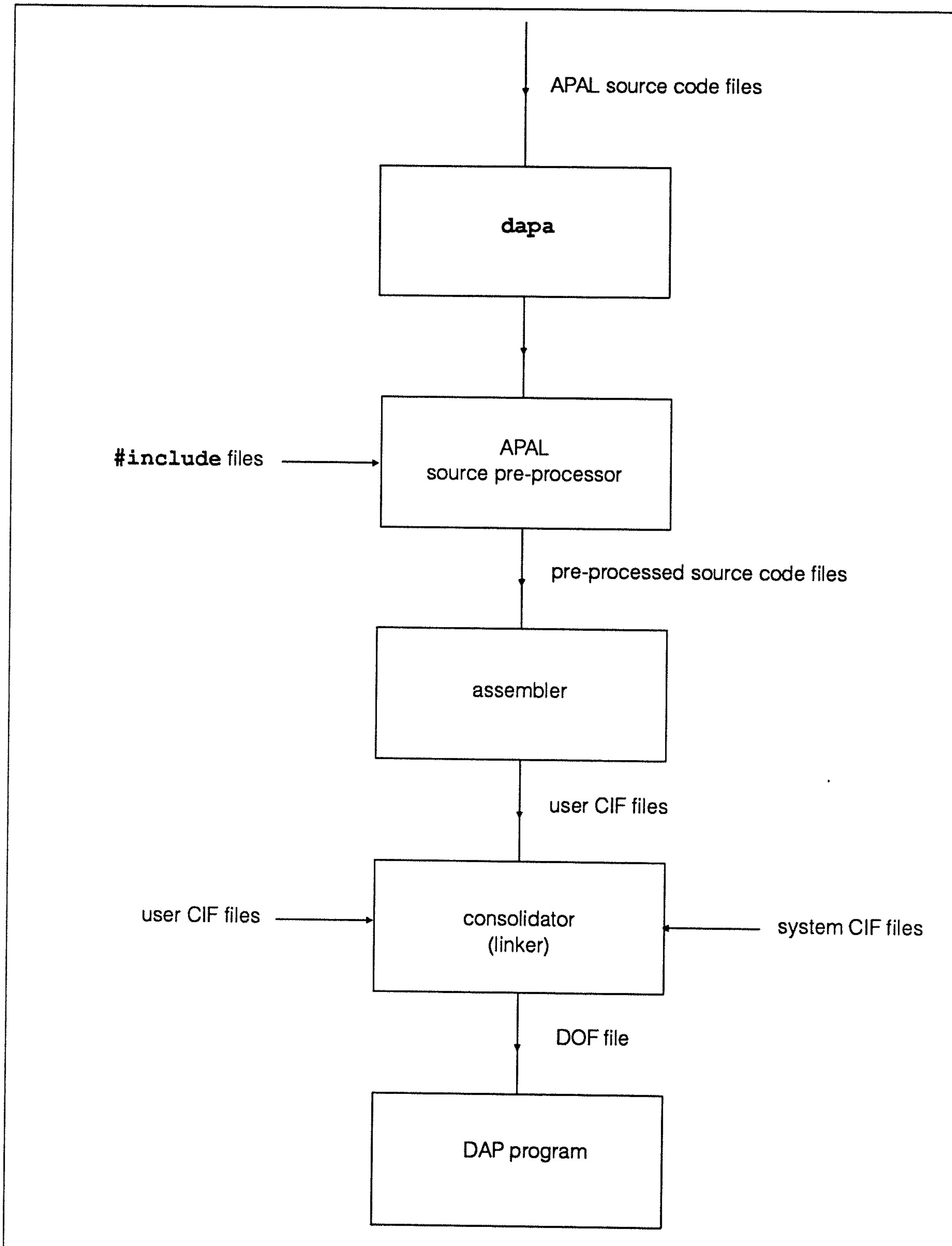


Figure 6.1 APAL assembly system

You cannot mix code for different DAP edge sizes in the same DAP program. However, you can mix code for DAPs with and without co-processor, but the resultant DAP program will only run on a DAP with co-processors, and the code compiled for a DAP without co-processors will not use the co-processors.

6.2 Components of the APAL assembly system

The APAL assembly process can be divided into 3 phases:

- Preprocessing
- Assembling
- Consolidating (linking)

The structure of the system is shown in figure 6.1 opposite. The command **dapa** controls all 3 phases and in the simplest case a single APAL source file is preprocessed, assembled and linked to form an executable DOF file. For example the command:

```
dapa  testprog.da
```

assembles the APAL source file and generates a DOF file with default name **d.out**.

The preprocessor phase expands tab characters, caters for any included files in the APAL source files and lets source lines be selected or ignored depending on the edge-size of the target DAP. The assembly phase generates output files in consolidator input format (CIF files), one CIF file for each input file. The CIF files are then passed to the consolidator, and linked together to form a DOF file. Options in the APAL assembly system are controlled by flags to the **dapa** command, as described in later sections.

6.3 APAL preprocessor

The APAL preprocessor takes the source files you input, and produces one continuous stream of output, which is passed to the assembler. It interprets *directives* in the source files, modifies tab characters and lets the assembler report errors by filename and line number. A directive always has a # in column one, and can be one of the following:

- **#include**
- **#if** or **#endif**

6.3.1 Tab characters

APAL source files can contain tab characters. Each tab character is replaced by the necessary number of spaces to make sure that the next character after the tab occurs in a column whose number is a multiple of 8, the first column being numbered 0. The expanded source code lines should not be longer than 80 characters; if it is an asembler error occurs, and you'll get a 'Line too long' message.

6.3.2 #include directive

The APAL preprocessor can handle source files containing one or more included files. The rules governing the use of the **#include** directive are the same as those for FORTRAN-PLUS (see section 2.3.2 page 10).

6.3.3 #if and #endif directives

You can select or ignore lines from the source code files by using the pair of directives **#if** and **#endif**. The form and working of the **#if** and **#endif** directives is the same as in the FORTRAN-PLUS preprocessor (see section 2.3.3, on page 14).

6.3.4 Preprocessor errors

The APAL preprocessor, **dapapp**, outputs diagnostic messages on the standard error stream. The error messages are self-explanatory.

6.4 APAL assembler

6.4.1 Assembler input and output

The APAL assembler is called automatically after the preprocessing phase, and generates output in consolidator input format. By default one CIF file is created for each input source file. The output file has the same name as the input file but with the file extension **.dc** instead of **.da**. You can have all the CIF files combined into a single file by using the **-j** flag. For example:

```
dapa  -j  cif  a.da  b.da
```

would combine the CIF output normally placed in **a.dc** and **b.dc** into one file **cif.dc**. The DOF file created by the above command will have the default name **d.out**, but you can change the name using the **-o** flag. Hence:

```
dapa  -o  dof  a.da
```

would create the CIF file **a.dc** and link it to produce the DOF file **dof**.

suppressing the linking phase

You can suppress the linking phase by specifying the **-c** flag. In this case the CIF files are produced but no DOF file is generated. You can link the CIF files to form a DOF file at a later time by using another flag to **dapa**, specifying the CIF files themselves as input. This means that if the APAL source is in several files, only those which have changed need to be reassembled. For example, if a program consists of two APAL source files, **a.da** and **b.da**, and if **a.dc** has been created earlier, then the command:

```
dapa  b.da  a.dc
```

would assemble **b.da** (to generate **b.dc**) and then link **a.dc** and **b.dc** to produce **d.out**.

In fact, you don't need to specify any APAL source files when using **dapa**. If all input files are CIF files, then the assembler is not invoked and the consolidator is entered immediately to link all the CIF files into a DOF file. For example:

```
    dapa  a.dc  b.dc
```

will simply link the files **a.dc** and **b.dc** into the default DOF file **d.out**.

6.4.2 Assembler listing and messages

By default the APAL assembler will not generate any assembly listings. However, you can use the **-L** flag to produce a brief, standard, or full source listing, according to its argument:

1 Brief

2 Standard

3 Full

You can also use the **-a** flag to obtain a cross-reference and attribute listing, and the **-e** flag to obtain an external reference and section listing. In all cases the listing is sent to standard output

6.4.2.1 Source listings

A source listing contains a line marking the start of each source file and a line reporting the creation of each CIF file (if appropriate). At the end of each module is a summary of the number of original source lines assembled. This number does not include those for macro expansion lines or lines input via **#include** directives.

A standard source listing is given in figure 6.2 on the next page.

parts of each line in the listing

Each line of the source listing occupies up to 112 columns and is divided into a number of fields, as discussed below:.

sequence numbers

- Sequence number – lined up under the word `source` in the third line of the listing below.

  Each line of APAL source code which is listed by the assembler is given a sequence number. This sequence number begins at one for each module and is used for cross reference purposes.

line type

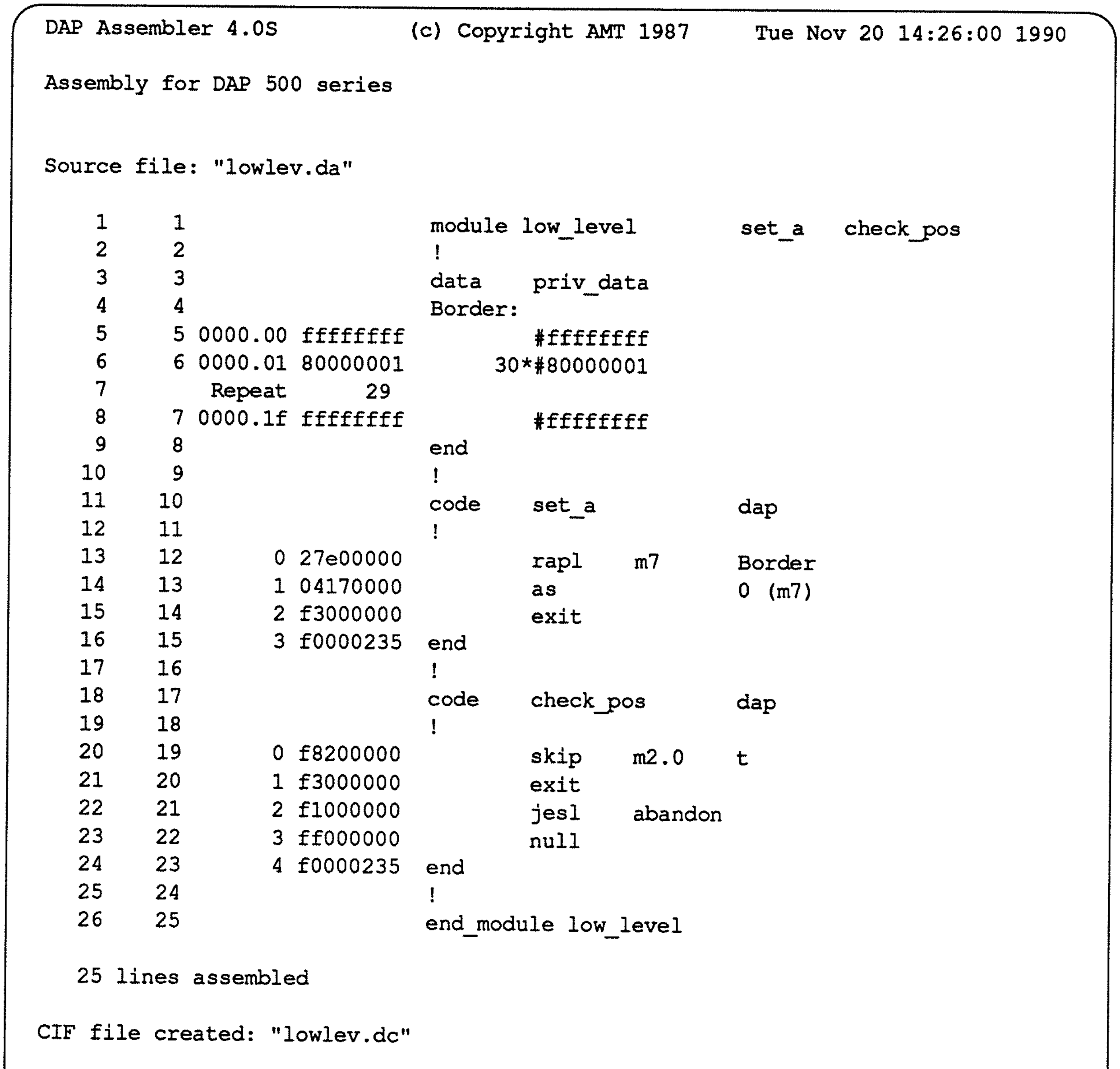
- Line type – none shown in the listing below, but would appear lined up under the space between `source` and `files` in the third line.

  The line type can take one of three forms depending on the origin of the source line:

  - A line produced as the result of macro expansion has the character **m**.
  - A line produced as the result of a substitution has the character **s** .
  - All other lines have a space character.

line number

- Line number – lined up under the word `file` in the third line of the listing below.



```
DAP Assembler 4.0S          (c) Copyright AMT 1987     Tue Nov 20 14:26:00 1990

Assembly for DAP 500 series


Source file: "lowlev.da"

    1     1                      module low_level        set_a    check_pos
    2     2                      !
    3     3                      data    priv_data
    4     4                      Border:
    5     5 0000.00 ffffffff             #ffffffff
    6     6 0000.01 80000001          30*#80000001
    7        Repeat      29
    8     7 0000.1f ffffffff             #ffffffff
    9     8                      end
   10     9                      !
   11    10                      code    set_a           dap
   12    11                      !
   13    12       0 27e00000             rapl    m7      Border
   14    13       1 04170000             as              0 (m7)
   15    14       2 f3000000             exit
   16    15       3 f0000235     end
   17    16                      !
   18    17                      code    check_pos       dap
   19    18                      !
   20    19       0 f8200000             skip    m2.0    t
   21    20       1 f3000000             exit
   22    21       2 f1000000             jesl    abandon
   23    22       3 ff000000             null
   24    23       4 f0000235     end
   25    24                      !
   26    25                      end_module low_level

   25 lines assembled

CIF file created: "lowlev.dc"
```

Figure 6.2 An example of a standard source listing

The line number can be derived in one of two ways, depending on the origin of the source line:

- Each source line read by the assembler is given a line number, beginning at one for the first record in each file (irrespective of whether or not the line is subsequently listed).

  The line number can therefore be used for editing purposes.

- During the expansion of a macro, each line of the macro body (irrespective of whether or not it is listed) is given a line number, beginning at one for the first line in the macro.

Such a line number identifies the line relative to the start of the corresponding macro definition. For nested macro calls, the line number begins again at one for each macro and assumes its original value on exit from each macro.

As there were no macros in the code listed above, all the line numbers form a single continuous sequence.

address

- Address – lined up under `DAP 500` in the second line of the listing above.

  The address is given, relative to the start of the corresponding data or code section, of each line that generates binary output (for example, a data declaration or an APAL instruction). The address can take one of the forms:

  - *pppp.ww* – as shown in lines 5, 6 and 7 of the listing above.

    For data values, where *pppp* is the plane address and *ww* is the word address; both values are in hexadecimal. This form of address only occurs with source for DAP 500 programs.

  - *pppp.rr.w* – no examples in the listing above.

    For data values, where *pppp* is the plane address, *rr* is the row address and *w* is the word address; the values are in hexadecimal. This form only occurs in programs for target DAPs of edge-size larger than 32, such as the DAP 600.

  - *wwwww* – as shown in lines 12 – 15, and 19 – 23 of the listing above.

    For instructions, where *wwwww* is the word displacement of the instruction from the start of the code section (in hexadecimal).

value

- Value – lined up under `Series` in the second line of the listing above.

  Each line that generates binary output has the value of the binary printed in hexadecimal.

  If a data declaration consists of one of the directives **`WORD`**, **`ROW`**, **`ALIGN`** or **`PLANE`** the value field contains:

  `xxxxxxxx`

repeated data items

If a data declaration specifies more than one data item, the value of each item is listed on a separate line together with its address. If a repeated data item is declared, as is the case on lines 6 and 7 in the listing above, the value is listed once, and the address and value field of the following line contains **`Repeat`** *n*, where *n*+1 is the value of the repeat count.

structured sequence of repeated data items

Had the data items constituted a structured sequence, and been declared with a repeat count, again only one instance of the structure would have been listed, and as before would have been followed by a line with **`Repeat`** *n* in the address and value field, where *n*+1 would have been the value of the repeat count. Additonally, the sequence would have been preceded by a line containing:

```
Begin level m
```

and followed by a line containing:

```
End level m
```

where *m* would have been an integer denoting the level of nesting of the structured sequence. These **`Begin`** and **`End`** lines would have been indented, so as to reflect the nesting of the structured sequence.

generating a literal

If an APAL instruction generates a literal value (for example, the instruction **`rlit`** where the value is too long to be loaded into the instructions itself by an **`rh or rhn`** instruction), the generated literal is printed in the value field on the following line, with a blank address field. The instruction address, the binary value generated by the instruction, and so on are given as usual.

source line

- Source line – lined up under `Copyright AMT` ... on line 1 of the listing above.

The source line as input to the assembler, subject to the listing level currently in force, and detailed below:

	Type of statement	*Assembler listing effect*		
		Full	*Standard*	*Brief*
Statement in the main body of the source code	Macro definition	Listed	Listed	Listed
	Macro call line	Listed before, during and after substitutions	Listed before and after substitutions	Listed after substitutions
	APAL source	Listed before, during and after substitutions	Listed before and after substitutions	Listed after substitutions
	Assembly directive	Listed before, during and after substitutions	Listed before and after substitutions	Not listed

	Type of statement	*Assembler listing effect*		
		Full	*Standard*	*Brief*
Statement generated by a macro call	Macro definition	Listed	Listed	Not listed
	Macro call line	Listed before, during and after substitutions	Not listed	Not listed
	APAL source	Listed before, during and after substitutions	Listed after substitutions	Not listed
	Assembly directive	Listed before, during and after substitutions	Not listed	Not listed

You can also control the listing level (brief, standard or full), by using the APAL LIST statement; see [3], the AMT publication *DAP Series: APAL Language*. The values **NONE**, **SHORT**, **SOURCE** and **FULL** correspond to **dapa**'s default, brief, standard and full listing levels respectively.

If assembly of the module produced diagnostics, you are also given:

- A record of the number of comments, warnings and errors.
- A list of the line numbers which generate diagnostics. This list uses the listing sequence numbers described above.

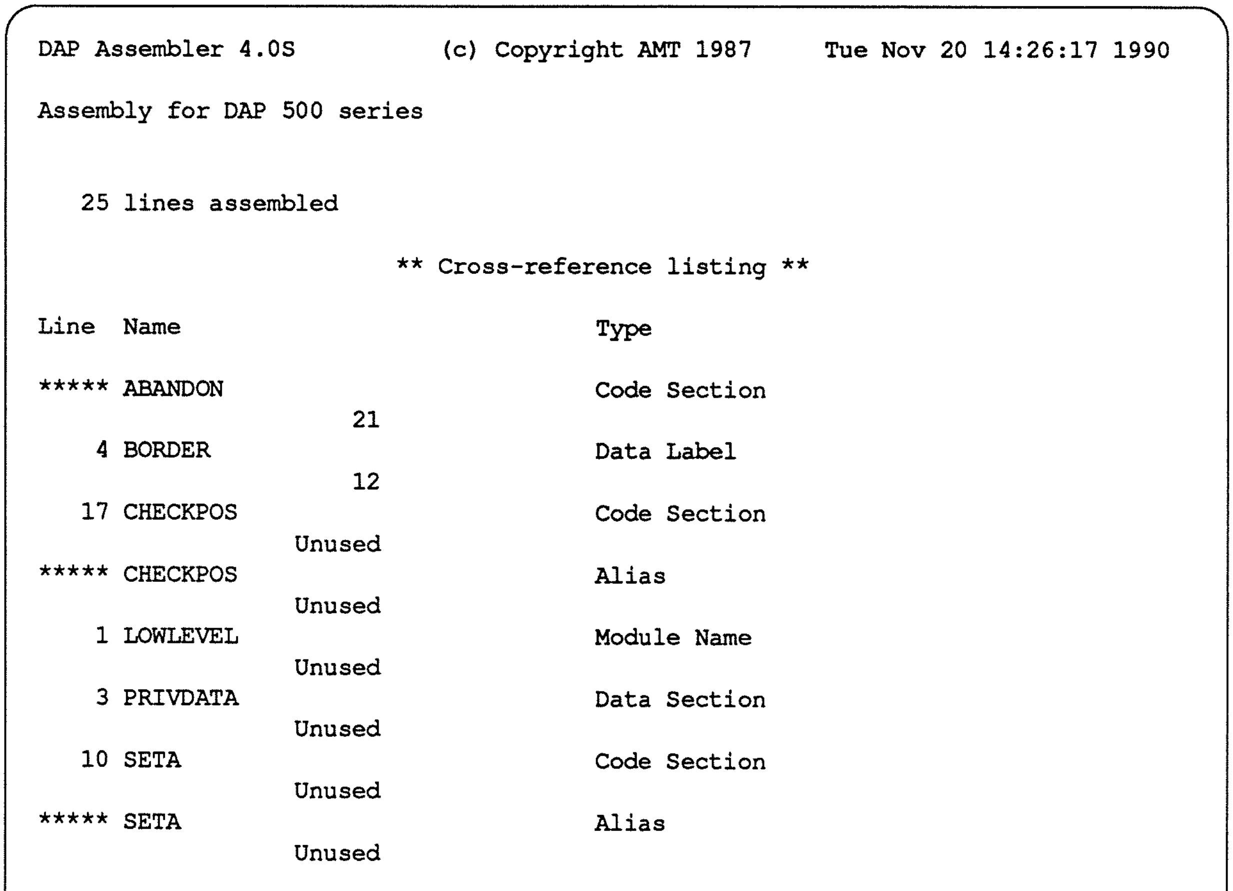
The name of the file containing the offending line(s) is displayed before the line(s) concerned. The rough place of an error in the line is shown by a ^ character. Whenever diagnostic messages are generated, a one line summary of the number of comments, warnings and errors is sent to the standard error stream. You can suppress the reporting of comments by specifying the **-q** flag to **dapa**.

6.4.2.2 Cross reference and attribute listing

A cross reference listing consists of information on each name declared or referenced in a module.

There are separate alphabetical lists for assembly-time variable names, macro names, and any other names. The names of macros declared outside the module are listed only if the name is referenced within the module. Macro variable names do not appear in the cross reference listing.

Figure 6.3 at the top of the next page gives an example of a cross reference and attribute listing.



```
DAP Assembler 4.0S         (c) Copyright AMT 1987     Tue Nov 20 14:26:17 1990

Assembly for DAP 500 series


   25 lines assembled

                         ** Cross-reference listing **

Line  Name                                Type

***** ABANDON                             Code Section
                  21
    4 BORDER                              Data Label
                  12
   17 CHECKPOS                            Code Section
                Unused
***** CHECKPOS                            Alias
                Unused
    1 LOWLEVEL                            Module Name
                Unused
    3 PRIVDATA                            Data Section
                Unused
   10 SETA                                Code Section
                Unused
***** SETA                                Alias
                Unused
```

Figure 6.3 An example of a cross reference and attribute listing

Each line of a cross reference listing has the fields:

- The line number of the line in which the name is declared. If a name is an alias or is not declared within the module, this field is asterisk filled. The field is unused for macro names.

  Notice in figure 6.3 that the names for the two code sections appear twice – once as aliases for the module name (in line 1 of the code), and once when they are declared.

- The name of the item.
- The type of the name, which can be any of:

  Data section
  Code section
  Data label
  Code label
  Identity
  Module name
  Alias
  Entry
  Macro name
  ATV

- A list of line numbers of the lines in which the name is referenced, excluding the declaring reference.

If you ask for a source listing as well as a cross reference and attribute listing, the module summary lines (just the one line – `25 lines assembled` – in the example listing opposite) that are included in the source listing are not included in the cross reference and attribute listing. In addition, lines are referred to by their sequence numbers, rather than their line numbers. Any generated lines will make the line and sequence numbers differ; in the cross reference and attribute listing without a source listing, if any generated line is referred to, the line number of the last input source line is used.

6.4.2.3 External reference and section listing

An external reference and section listing is an alphabetical list of all the sections in a module, with their sizes and attributes, followed by an alphabetical list of all the references in the module assumed to be external.

Figure 6.4 below gives a example of an external reference and section listing.

```
DAP Assembler 4.0S           (c) Copyright AMT 1987     Tue Nov 20 14:26:31 1990

Assembly for DAP 500 series


   25 lines assembled


                                 ** Section list **

Name                              Properties

ABANDON                           Code External
CHECKPOS                          Code Size:     5 words DAP
PRIVDATA                          Data Size:    32  rows Private
SETA                              Code Size:     4 words DAP

                              ** External references **

ABANDON
```

Figure 6.4 An example of an external reference and section listing

If you ask for a source listing as well as a cross reference and attribute listing, the module summary lines (just the one line – `25 lines assembled` – in the example listing opposite) that are included in the source listing are not included in the cross reference and attribute listing. In addition, lines are referred to by their sequence numbers, rather than their line

numbers. Any generated lines will make the line and sequence numbers differ; in the cross reference and attribute listing without a source listing, if any generated line is referred to, the line number of the last input source line is used.

6.4.2.4 Assembly diagnostics

Assembler diagnostic messages are classified according to their severity level.

There are four severity levels:

1 Comment

2 Warning

3 Error

4 Terminal error

These error messages are similar in meaning to those specified for the FORTRAN-PLUS compiler; see section 2.4.2.5 on page 19. The assembler error messages are self explanatory.

6.4.2.5 Defining assembly-time variables

The **-V** flag to **dapa** lets you define and initialise assembly-time variables for use in the APAL source. The form of the flag is:

-V *var-name=var-value*

You can have up to 10 **-V** flags in each invocation of **dapa**.

Having such flags is equivalent to having at the start of each APAL source file the statement:

VAR *var-name=var-value*

For details of how to use assembly time variables, see chapter 11 of the APAL manual.

6.4.2.6 Profiling APAL programs

When an APAL program is executing, profiling information is generated, provided that you assembled the APAL source using the **-p** flag to **dapa**, and that you **#include**d the AMT system macros in file **amtmacs.da** in every APAL module.

Most APAL programs make use of the AMT system macros, which used to be held in file **usrmacs.da**. **usrmacs.da** is still available, but AMT has upgraded the macros and put them in **amtmacs.da**, and it is these upgraded macros that are needed when profiling information is generated.

If you don't already use the sytem macros in your APAL code, then profiling information will be generated for every module that has at its start the statement:

```
#include amtmacs.da
```

You will also need to adopt the entry and exit conventions described in [3], *DAP Series: APAL Language.*

If you already have **#include usrmacs.da** in your code, all you need to do is to change the **usermacs.da** to **amtmacs.da**.

Profiling information is also generated for FORTRAN-PLUS program, provided you use the **-p** flag to **dapf** when you are compiling them. Mixed programs (see section 6.5 below) will generate profiling information, provided the different sections were assembled or compiled as described above.

The profiling information is stored in file **dmon.out** in your current directory when the program is run. You can use the utility **dapprof** to analyse the file; **dapprof** is described in section 3.5 on page 54. When you don't need the profiling information anymore, you should re-assemble or re-compile without the **-p** flag.

6.4.3 Assembly of APAL trace statements

All APAL **trace** statements have an associated level number which you can use to control their execution at run-time (see section 3.4.1 on page 44). The level number is also used to control the conditional assembly of the **trace** statements. The **-t** flag specifies the maximum level number of **trace** statements which are to be assembled. **-t** can take any value from 0 to 15, 15 meaning all **trace** statements are assembled and zero meaning no **trace** statements are assembled.

For example:

```
dapa  -t2  a.da
```

will assemble all **trace** statements at levels 1 and 2. The default value for the **-t** flag is zero.

6.5 Mixing FORTRAN-PLUS and APAL routines

You might sometimes want to create a DAP program which is a mixture of both FORTRAN-PLUS and APAL routines. To do this is straightforward because the format of the consolidator input files is independent of the source language, and the linking phase of **dapf** is identical to that of **dapa**. Therefore CIF files created by **dapf** can be used as input to **dapa** and vice versa.

As an example consider a program consisting of the FORTRAN-PLUS file **f.df** and the APAL file **a.da**. To assemble and link these together the following commands could be used:

```
dapf  -c  f.df
dapa  f.dc  a.da
```

Alternatively, the commands:

```
dapa  -c  a.da
dapf  a.dc  f.df
```

could be used. Note the use of the **-c** flag in the first command to suppress the linking phase after the generation of the CIF files.

For details of the requirements for an APAL code section that is to communicate with a FORTRAN-PLUS procedure, see chapter 9 of [3], *DAP Series: APAL Language.*

6.6 APAL linking

The **dapa** command invokes the APAL consolidator (linker) as the last phase of the assembly process. In fact there is no difference between the APAL consolidator and the FORTRAN-PLUS consolidator, and so you can refer to section 2.5 on page 24 on the FORTRAN-PLUS consolidator for details of linking in CIF library files, consolidator maps, messages and diagnostics, and for examples.

6.7 dapa flags

This section contains a summary of all the flags available with **dapa**. **dapa** flags and filenames can appear in any order, but the consolidator searches files and CIF libraries in the sequence specified and this can be significant (see section 5.4 on page 108).

-a Generate a cross reference and attribute listing.

-c Do not link.

-e Generate an external reference and section listing.

-I *dirname* Modify search paths for **#include** files. This option instructs the preprocessor to add *dirname* to the search path for **#include** files whose names do not begin with a **/** .

-j *name* Join all CIF into one file called *name*.**dc** .

-l *name* Pass the CIF library associated with the package *name* to the consolidator.

-L *n* Generate a source listing to the level specified by *n*. Valid values for *n* are:

1 brief listing

2 standard listing

3 full listing

-m*n* Generate a consolidator map to the level specified by *n*. Valid values for *n* are:

1 brief map

2 standard map

3 full map

-o *filename* Generate a DOF file called *filename* instead of the default name **d.out** .

-p Generate profiling information for every module for which **amtmacs.da** is **#include**d.

-q Suppress assembler comment messages.

-s*n* Set DOF stack record to *n* planes.

-s+*n* Set DOF stack record to *n* planes plus the consolidator estimate.

-t*n* Assemble source **trace** statements which have a level less than or equal to *n*. Valid values for *n* are 0 to 15 inclusive. The default value is 0.

-V *var-name=var-value*

Define the assembly-time variable *var-name* with value *var-value*

-y Inhibit the production of CIF files. The consolidator is not run. This option is in effect a syntax checker.

Other flags are ignored and a warning message is produced. If conflicting options are specified (such as **-L2 -L3**), the last one is used and the previous ones ignored.

Chapter 7

Controlling multi-programming on the DAP

7.1 Introduction

All models of the DAP are capable of running up to 29 user programs at the same time. The actual number loaded at any one time depends on each program's store requirements.

Programs resident in the DAP are run on a round-robin basis, with a change in the current process occuring after an adjustable time period, or when the current process is suspended for any reason. As far as the DAP is concerned, each program running in the machine constitutes one process.

Users need take no action to use this multi-programming facility, and do not need to know how many other processes are running on the DAP at the same time as theirs. Currently there are no inter-process communication facilties on the DAP. However, facilities exist on the DAP to let people with suitable host system privileges control the flow of work in the machine. This chapter describes those faciltiies.

7.1.1 Definition of terms

The following terms are used in this chapter:

DAP Process ID

- A DAP program is allocated a *DAP Process ID* when you try to load it.

  The owner of a DAP process is that user whose user id was effective when **DAPCON** was called in the associated host program.

slot time

- The *slot time*, a measure of the maximum time for which a particular DAP program can run without system interruption when it is the active process, is set by the product of the 3 factors:

  timeslice

  - The current value of *timeslice*, the same for all DAP processes running in the machine at a given time. It is the 'unit' of processing time or *milltime* that is allocated to user processes – the larger the value, the greater the slot time for each process.

  priority

  - The process *priority*, the priority given to a particular process; the higher the value of *priority*, the higher the priority of the process. You can vary each process's priority and hence its slot time, and so allocate to different processes a different proportion of the DAP time available.

- A factor dependent on the version of DAP software in use.

7.2 Controlling DAP programs

Since up to 29 programs can be resident in the DAP at any one time, you might want some way of controlling the running of DAP processes remotely. For example, you might want to suspend all but one process temporarily in order to run a demonstration.

To satisfy such requirements, two files are supplied with the DAP basic software. One file contains:

- A library of compiled low-level subroutines which pass process control messages to and from the DAP. You are able to link your own command line interpreter to this library, as a front-end tailored to your requirements.

The other file contains:

- A fully compiled program which has a very simple example interpreter built onto the library, giving you a guide to what is required from an interpreter.

7.3 Monitoring usage

A record is kept in a file on the host of all programs submitted to the DAP, along with details of their execution times. This file, **`/usr/adm/dapsyslog`**, is opened and written to by the **`dapboot`** process, and entries are made in it each time a program is unloaded from the DAP.

The following information is recorded in the file:

- The user name
- The DAP program name
- The time the system loaded the program
- The time the system unloaded the program
- The total DAP milltime used by the program
- The priority that was current when the program was unloaded

In addition, entries are also made in the file whenever the value of timeslice is changed.

All users have read access to this file and each time **`dapboot`** is invoked it opens the file in append mode. If you are a system manager, you may find it useful to reduce the size of the file periodically by editing out some of the earlier entries, to save disk space.

7.3.1 Facilities available

The low-level library supplied with the DAP basic software provides the system manager and users with routines to:

- Suspend a DAP process.
- Restart a DAP process.
- Print information about one particular DAP process, or about all DAP processes.
- Kill a DAP process.
- Set the priority of a DAP process.
- Set the value of the system timeslice.

The routines use a reserved channel to communicate with the DAP, so there is no danger of their being unable to gain access to the DAP, even when the DAP is being heavily used. You access these routines via a suitable interface that you can tailor to your own requirements; **`dapoip`** is a program which includes a simple example of an AMT-written interface, and is described in section 7.3.3 on page 131.

Any number of interface programs using these routines can be running at the same time, but an error will be reported if the **`dapboot`** process is not already running when a routine is called.

7.3.2 Specification of the routines

The low-level routines provided for you to control your multi-programming environment are held in the file **`/usr/lib/dap/dapcontrol.o`**.

The specifications of the routines are:

- **`void priority`**`(`*proc-id*`,` *prior*`)`
  `short` *proc-id*`,` *prior*`;`

  The priority of process *proc-id* is set to *prior*, providing the current effective user (as defined by normal UNIX practice) is either the owner of *proc-id*, or is `root`.

- **`void timeslice`**`(`*ts*`)`
  `int` *ts*`;`

  If *ts* = 0, then the current value of the timeslice is sent to your standard output channel (usually the host screen); otherwise the timeslice is set to *ts*.

- **`void list`**`(`*proc-id*`)`
  `short` *proc-id*`;`

  If *proc-id* = 0, then information on all DAP processes is sent to standard output, otherwise information on process *proc-id* is sent. The current value of the system timeslice is also sent.

The information is given under the following headings:

DAPID – The DAP process ID

HPID – The associated host process ID

Dev – The minor device number the process has open

Status – One of :

Idle
Queued
Loading
Sus'd (for suspended)
Running
Unloading
Unloaded

Pri'ty – The priority of the process

Milltime – The total DAP milltime used by the process, in milliseconds

S_state – If the process is suspended (or if it will be as soon as it is fully loaded), **S_state** will be some combination of:

R – returned to the host program
P – paused (or is in some diagnostic mode)
H – halted and dumping after a signal
S – suspended by root
s – suspended by the owner
W – awaiting the next timeslice
a – opening a host file
l – seeking within a host file
t – establishing current position within a host file
d – transferring data to or from a host file
r – closing a host file
F – using the fast input and output channel (I/O)
V – using the VME

Only when nothing is set in this **S_state** field is a program actually executing

Username – The owner of the process

Dofname – The name of the DOF file containing the DAP program

- **void dapkill**(*proc-id*)
  short *proc-id;*

  If the current effective user either is the owner of DAP process *proc-id*, or is **root**, then that process and its associated host program are killed.

- **`void suspend`**(*proc-id*)
  `short` *proc-id;*

  If the current effective user either is the owner of DAP process *proc-id*, or is **`root`**, then that process will become suspended by the owner or by **`root`** respectively.

- **`void restart`**(*proc-id*)
  `short` *proc-id;*

  If the current effective user either is the owner of DAP process *proc-id*, or is **`root`**, and that process has been suspended by that user, then the suspension is lifted.

7.3.3 Example interface

The command **`dapoip`** invokes a simple example interface built onto the routines described above. Once invoked, it continually asks for commands by displaying its prompt:

`dapoip:`

The commands available are:

`h` Print this help text.

`k` *n* Kill process *n* .

`l` [*n*] List the status of process *n*, or of all DAP processes if *n* is absent.

`p` *n m* Set priority of process *n* to *m*. The command is only valid if it is issued by the owner of the DAP process *n* (who can set priority to a value within the range 1–5), or **`root`** (who can set priority within the range 1-10).

`q` Leave **`dapoip`** .

`r` *n* Resume process *n* .

`s` *n* Suspend process *n* .

`t` [*n*] Set system timeslice to *n*. If *n* is 0 or absent, show the current value of timeslice. *t* can only be changed by **`root`**, who can set it to a value in the range 1-255.

To illustrate the sort of display you might get, the simple example program used in chapter 4 was run on DAP hardware (on which two other DAP programs were already running), and **`dapoip`** started. The display produced was:

```
host%
dapoip: l
DAPID HPID  Dev Status  Pri'ty  Milltime   S_state   Username  Dofname

  510 26209   3 Sus'd     5       299633   RW        djh       rippledap
  517 26233   2 Queued    5            0             sjh       testmat7
  518 26291   4 Sus'd     5            0   PW        asb       dapobj

Timeslice: 10
dapoip:
```

Appendix A

Command specification

A.1 dapa

This section contains a summary of all the flags available with **dapa**. **dapa** flags and filenames can appear in any order, but the consolidator searches files and CIF libraries in the sequence specified and this might be significant (see section 5.4 on page 108).

-a Generate a cross reference and attribute listing.

-c Do not link.

-e Generate an external reference and section listing.

-I *dirname* Modify search paths for **#include** files. This option instructs the preprocessor to add *dirname* to the search path for **#include** files whose names do not begin with a **/** .

-j *name* Join all CIF into one file called *name*.**dc** .

-l *name* Pass the CIF library associated with the package *name* to the consolidator.

-L*n* Generate a source listing to the level specified by *n*. Valid values for *n* are:

1 brief listing

2 standard listing

3 full listing

-m*n* Generate a consolidator map to the level specified by *n*. Valid values for *n* are:

1 Brief map

2 Standard map

3 Full map

-o *filename* Generate a DOF file called *filename* instead of the default name **d.out** .

-p Generate profiling information when the program is run.

-q Suppress assembler comment messages.

-s*n* Set DOF stack record to *n* planes.

-s+*n* Set DOF stack record to *n* planes plus the consolidator estimate.

-t*n* Assemble source **trace** statements which have a level less than or equal to *n*. Valid values for *n* are 0 to 15 inclusive. The default value is 0.

-V *var-name=var-value*

Define the assembly-time variable *var-name* with value *var-value*

-y Inhibit the production of CIF files. The consolidator is not run. This option is in effect a syntax checker.

Other flags are ignored and a warning message is produced. If conflicting options are specified (such as **-L2 -L3**), the last one is used and the previous ones ignored.

A.2 daped

This section summarises all the **daped** commands. **daped**'s main use is to adust the size of the stack for your DAP program, but it has other uses.

d [*section-name*]

Delete the named inserted data section.

If you don't supply a *section-name*, **daped** will prompt you for one.

Inserted data sections are specially marked in DOF files, so you can delete sections inserted in previous runs of **daped** .

f [*DOF-file-name*]

Select the named new DOF file.

If you don't supply a *DOF-file-name*, **daped** will prompt you for one.

If you have made changes to the current file since you last saved it (with a **w**) you will see a warning when you issue an **f**, and your requested DOF file is not selected. If you then issue another **f** before you make any more changes, the requested DOF file is selected, and any changes you made to the current file since you last saved it will be lost.

h

Print a brief help text.

i [*section-name*]

Insert the named new data section.

If you don't supply a *section-name*, **daped** will prompt you for one. In either case you are then prompted for the required size of the section.

Creates a new array store section just before the stack section. There is no way of initialising the newly-created section of store. One use of such a section is as a buffer area in MCUCP (the MCU control program) for device drivers.

l

List the contents of the currently-selected DOF file.

The listing is of the current state of the file, after any editing you have carried out, and whether or not you have saved the file since changes were made. The output is similar to a consolidator map or to that provided by the **map** command in **psam** .

q

Leave **daped** .

If you have made changes to the current file since you last saved it (with a **w**) you will see a warning when you issue a **q**, and you will not exit **daped**. If you then issue another **q** before you make any more changes, you will exit **daped** and any changes you made to the current file since you last saved it will be lost.

`r`

Restore the original array store map.

This command restores the current DOF file to the state as originally created by the consolidator: inserted data sections are removed and the stack allocation reverts to its original size.

`s` [*n*]

Adjust the size of the stack section to be *n* DAP planes. If you omit *n*, the current size is displayed, in planes, and you are prompted for the required new size.

`w` [*DOF-file-name*]

Save the current DOF file to the file *DOF-file-name.*

If you don't supply *DOF-file-name*, then the name of the current file is used.

A.3 `dapf`

This section contains a summary of all the flags available with **`dapf`**. **`dapf`** flags and filenames can appear in any order, but the consolidator searches files and CIF libraries in the sequence you specify – and the order can be significant (see section 5.4 on page 108).

`-a` Generate a cross reference and attribute listing

`-c` Do not link

`-D`*n* Generate various levels of diagnostic information that might be used in the event of run-time errors or by **`dapdb`**. Valid values for *n* are 0 to 2 inclusive; it controls the extent of available information:

Value of n	*Effect*
0	Subprogram names only are available.
1	As for 0, plus line numbers.
2	As for 1, plus names and values of all variables in common areas. or currently on the stack..

The default value is 2.

`-e` Generate an external reference listing

`-g` Allow single-stepping (execution of one line of source code) from within **`psam`**.

`-I` *dirname* Modify search paths for **`#include`** files. This option instructs the preprocessor to add *dirname* to the search path for **`#include`** files whose names do not begin with **/** .

`-j` *name* Join all CIF files into one file called *name*`.dc`

`-l` *name* Pass the CIF library associated with the software called *name* to the consolidator.

`-L`*n* Generate a source listing of the level specified by *n*. Valid values for *n* are:

Value of n	*Effect*
1	Brief listing
2	Full listing

By default, no listing is given.

`-m`*n* Generate a consolidator map of the level specified by *n*. Valid values for *n* are:

Value of n	*Effect*
1	Brief map
2	Standard map
3	Full map

By default, no map is given.

`-o` *filename* Generate an executable DAP program file called *filename* instead of the default name **`d.out`** .

`-O`*n* Carry out the optimisations specified by *n*. Valid values for *n* are:

Value of n	*Effect*
`0`	No optimisation.
`1`	MCU registers and co-processor memory are optimised using simple cacheing.
`2`	As 1, plus expression analysis, to optimise co-processor usage.

If *n* is omitted, the highest level of optimisation available in the release of the compiler being used is selected.

By default, no optimisations are carried out.

`-p` Generate profiling information when the program is run.

`-q` Suppress compiler comment messages.

`-r`*x* Suppress run-time checks in the program according to the value of *x*. Valid values for *x* are:

Value of x	*Effect*
`c`	No checking for the shape of operands in expressions for conformance.
`d`	No checking whether the value of the `do` loop increment is zero.
`n`	No checking of real data for normalisation before floating point operations are carried out.
`o`	No checking for overflow.
`p`	No checking if formal and actual parameters to routines conform in type, data-length, shape and mode.
`s`	No checking if subscripts are in range.
`a`	None of the above-mentioned checks are applied – that is, no run-time checks are applied.

By default, no checks are suppressed.

`+rh` Check if formal and actual parameters to routines match in their non-parallel dimensions.

`-s`*n* Set DOF stack record to *n* planes.

`-s+`*n* Set DOF stack record to *n* planes plus the consolidator estimate.

`-t`*n* Compile source `trace` statements which have a level less than or equal to *n*. Valid values for *n* are 0 to 5 inclusive. The default value is 0.

`-y` Inhibit the production of CIF files. The consolidator is not run. This option is in effect a syntax checker.

Other flags are ignored and a warning message is produced. If conflicting options are specified (such as `-L1 -L2`) the last one is used and the previous ones ignored.

A.4 daplib

This section contains a summary of all the **daplib** flags.

-D *syn=name* Define an additional synonym *syn* for an existing module with the synonym *name* .

-f *syn-exp* Filter out from the input files any module with a synonym matching *syn-exp*, and do not place it in the output library. The filter is applied to each input file in turn.

syn-exp can include one or more specific synonym names, and can include a combination of wild cards and regular expressions – for details, see section 5.2.2 on page 103.

-k Kill (remove) any second or subsequent occurrences of any synonyms in the input files. If a removed synoym is the only synonym of an input module, that module is not included in the output library, and a warning message is sent to standard output.

-L*n* Generate a **daplib** listing of the level specified by *n*. Valid values of *n* and the effects they have are:

1 Lists the output library only.

2 Lists each input file only.

3 Lists both input files and the output library.

-m *syn-exp* Only copy a module from an input file to the output library if the module has a synonym matching *syn-exp*.

syn-exp can include one or more specific synonym names, and can include a combination of wild cards and regular expressions – for details, see section 5.2.2 on page 103.

-o *name* Generate an output library called *name*.**dl** instead of the default name **daplib.dl** . If *name* ends with **.dl**, do not add a further **.dl** .

-x *syn-exp* Delete any synonym matching *syn-exp* from the output library. If *syn-exp* matches the only synonym of an input module, that module is not included in the output library, and a warning message is sent to standard output.

syn-exp can include one or more specific synonym names, and can include a combination of wild cards and regular expressions – for details, see section 5.2.2 on page 103.

-y Inhibit the production of an output library.

A.5 **dapopt**

This section contains a summary of all the **dapopt** flags.

-a*n* Ignore assembled APAL **TRACE** statements of level greater than *n*, where *n* is in the range 0 to 15, and has a default value of 15.

-b*n* When the DAP program is entered, take the action specified by *n*:

- **0** Start execution of the program.
- **1** Do not start execution of the program, but enter **psam** directly.

-d*n* Set the runtime diagnostics level to *n*, where *n* is in the range 0 to 2, and has a default value of 0.

-D *name* Send diagnostics to file *name* .

-D Send diagnostics to the standard error channel.

-e*x* If a run-time error occurs, take the action specified by *x*. Valid values of *x*, and the resultant action:

- **a** Abort.
- **c** Continue.
- **p** Enter **psam** .
- **dc** or **cd** Dump and continue.

The default is *x* = **p** — drop into **psam** .

-f*n* Ignore compiled FORTRAN-PLUS **TRACE** statements of level greater than *n*, where *n* is in the range 0 to 5, and has a default value of 5.

-h*n* Generate a histogram based on a 'slice' of *n* instructions required (only affects simulator).

-h0 Do not generate a histogram.

-l*n* Set the histogram lower code address limit to *n* (in decimal, octal [prefix 0] or hexadecimal [prefix 0X]) (only affects simulator).

-L Generate a list of file options and send it to standard output.

-o *name* Send the output DOF to file *name* .

-q Suppress **dapopt** comments output.

-S *name* Send statistics to file *name* (simulator only).

-S Send statistics to standard output.

-s1 Run the DAP program on the DAP simulator.

-s0 Run the DAP program on DAP hardware.

-t*n* Generate the specified timing information (simulator only). Valid values for *n*, and the information generated:

0	None
1	Standard
2	Full

-u*n* Set the histogram upper code address limit to *n* (in decimal, octal [prefix 0] or hexadecimal [prefix 0X]) (only affects simulator).

-x Reset all options to default values and ignore previous flags (if any).

-y Do not generate any DOF output.

A.6 Summary of **psam** and **dapdb** commands

Command	**psam**	**dapdb**
alias	Create alternative name(s) for **psam** or **dapdb** commands.	
array	Display the contents of an area of array store.	
attributes	Display the attributes of a variable.	
backtrack	Display details of the procedure(s) currently on the stack.	
breakpoints	Display the current breakpoint settings.	No such command.
clear	Clear breakpoints.	No such command.
code	Disassemble and display APAL object code from the current code section.	No such command.
continue	Exit psam and run the DAP program past the specified numer of breakpoints or **PAUSE** statements.	No such command.
core	No such command.	Change the current dump file to the one specified.
date	Display the current time and date.	
disable	Disable breakpoint(s).	No such command.
display	Display the contents of the specified FORTRAN-PLUS variables on entry to **psam** .	
down	Change the current procedure to the procedure which is one lower on the stack.	
dump	Copy the current DAP state to the dump file.	No such command.
echo	Display the arguments to this command.	
enable	Enable the specified breakpoint(s).	No such command.
errors	Display the positions of the cumulative errors in vectors and matrices of the same shape as any user-declared error recording masks; also display whether or not there have been any errors in variables of any shape.	
file	Change the current file to the one specified.	
help	Display a summary of the **psam** and **dapdb** commands.	
history	Display the commands used earlier in the current **psam** or **dapdb** session.	
list	Display part or all of the contents of the curent file.	

Command	**psam**	**dapdb**
macro	Execute **psam** or **dapdb** commands from the specified file.	
map	Display an occupancy map of either of the code or array stores, or both.	
masks	Display the current FORTRAN-PLUS user-defined error interrupt mask(s).	
message	Repeat the information displayed on entry to **psam** or **dapdb** .	
next	Step program execution through the specified number of FORTRAN-PLUS source statements, starting with the next statement, and treating any procedure calls as a single statement (*cf* **step**).	No such command.
print	Display the contents of the specified FORTRAN-PLUS variable(s) or component(s) from variable(s).	
procedure	Change the current procedure to the one specified.	
quit	Quit the current **psam** or **dapdb** session.	
registers	Display one or more of the MCU, edge and PE registers, the APAL carry and overflow flags, and the hardware DO loop iteration number.	
save	Save the current settings of **psam** or **dapdb** environment variables to file **.defaults** in the user's home directory.	
select	No such command.	Change the current DAP state dump to the one specified (which must be from the same dump file).
set	Set the specified **psam** or **dapdb** environment variable to the given value.	
status	Display the current breakpoint(s) in command format.	No such command.
step	Step program execution through the specified number of FORTRAN-PLUS source statements, starting with the next statement and treating each statement in a procedure call as one statement (*cf* **next**).	No such command.
stepi	Step program execution to the next APAL instruction.	No such command.
stop at	Set a breakpoint at the start of a FORTRAN-PLUS source statement.	No such command.

Command	**psam**	**dapdb**
stop in	Set a breakpoint on the first executable line of a FORTRAN-PLUS procedure.	No such command.
stopi at	Set a breakpoint at a given offset in an APAL procedure.	No such command.
stopi in	Set a breakpoint at the start of an APAL procedure.	No such command.
top	Change the current procedure to the procedure at the top of the stack.	
time	Display total execution time since the start of the DAP program, and execution time difference since the time command was last issued.	
unalias	Delete the specified alternate command name(s).	
unset	Unset the value of (that is, set to default) the specified **psam** or **dapdb** environment variable.	
up	Change the current procedure to the procedure which is one higher on the stack.	

Appendix B

Messages from the run-time system

All the messages are preceded by:

`Run-time Error:`

The messages are:

`Array store full`

Possible cause: The program requires more stack space than is available on the DAP.

Action: Reduce the amount of space needed (for example, use fewer local variables, use **`equivalence`**, use lower precision, and so on), try consolidating with fewer stack planes (for details, see section 2.5.2.1 on page 25), or use **`daped`** (see section 2.5.2.2 on page 27).

`Attempted access outside array store datum or limit`

Possible cause: The program requires more stack. You will normally get this message when you are running an APAL program, but it can sometimes occur with small FORTRAN-PLUS programs. (An APAL program which attempts addressing with an MCU register containing an invalid address might also receive this message).

Action: Request more stack with the **`-s`** flag when consolidating (for details, see section 2.5.2.1 on page 25), or use **`daped`** (for details, see section 2.5.2.2 on page 27).

`Attempted access outside code store datum or limit`

Possible cause: The DOF file is probably corrupt.

Action: Reconsolidate the program.

`Cannot open file` *name*

Possible cause: The program does not have the required read or write access to the specified file.

Action: Check the file access permissions (set appropriate permissions with **`chmod`** *n filename*).

`Code and array store full`

Possible cause: The program is too big for the DAP, both in array and code store requirements.

Action: Reduce array store requirements by lowering stack request and make sure that redundant routines are not consolidated.

`Code store full`

Possible cause: The code of the DAP program is too large for the available code store.

Action: Reduce the program code size or install more code store; check that redundant routines are not being consolidated (use **`-m 3`** with **`dapf`** or **`dapa`**) - if so try using CIF libraries (for more details, see chapter 5, starting on page 101).

`Coprocessor store full`

Possible cause: The program is too large to fit into the co-processor store.

Action: Reduce the size of your program if possible.

`Data section` *name* **`unrecognised by dapsen or daprec`**

Possible cause: A non-existent **`common`** block name has been passed to **`dapsen`** or **`daprec`**.

Action: Check the spelling of *name*.

`DOF file does not match edge-size of DAP.`

Possible cause: DOF file has been compiled/assembled for the wrong size of DAP.

Action: Set environment variable **`DAPSIZE`** correctly (for more details, see section 2.1 on page 7) and then recompile or re-assemble the DOF file.

`DOF file requires DAP with co-processor`

Possible cause: You compiled and/or linked your program with **`DAPCP8`** set to **`yes`**, but the DAP you are trying to run your program on does not have co-processors.

Action: Use **`unsetenv DAPCP8`**, then compile and link your program again.

`Entry point` *name* **`unrecognised by dapent`**

Possible cause: The name passed to **`dapent`** is not an entry subroutine name in the DOF file loaded by **`dapcon`**.

Action: Check the correct DOF file is loaded, the spelling of *name* and the declaration in the DAP program.

`Illegal instruction in DO loop`

Possible cause: The DOF file is corrupt.

Action: Relink the program.

`Illegal or undefined instruction`

Possible cause: The DOF file is corrupt.

Action: Relink the program.

`Load failed`

Possible cause: This message normally appears after a previous error.

Action: Refer to previous error for further information.

`Not a DOF file`

Possible cause: The file specified to **dapcon** does not contain valid DOF.

Action: Relink the DAP program; check the name given to **dapcon** is correct.

`Not connected to any DAP process`

Possible cause: A call to **dapent, dapsen** or **daprec** has been made after a call to **dapcon** failed.

Action: Check the response from **dapcon** (for more details, see section 3.4.1 on page 37).

Either of:

`Privileged instruction in user code` *or*
`Using MP in user mode`

Possible cause: These errors should not occur.

Action: Contact your AMT representative.

`Transfer request too large for data section` *name* **`when executing dapsen or daprec`**

Possible cause: The number of bytes requested in **dapsen** or **daprec** is greater than the space allocated in the **common** block.

Action: Check the size requested matches the data declarations.

`Unable to read from DOF file`

Possible cause: **dapcon** does not have read access to the DOF file.

Action: Change the permissions to include read (to change permissions, use **chmod** *n filename*).

The following messages are preceded by:

`Warning:`

`No free DAP resources`

Possible cause: **dapcon** failed due to the DAP being fully used already.

Action: Check the response from **dapcon** in the host program and try again.

`No dump taken program still loading`

Possible cause: The key combination <CONTROL-\> was pressed (to get a dump of the DAP state) when the DAP program was still loading.

Action: Let the loading of the DAP program complete before you press <CONTROL-\> to get a dump of the DAP state.

Either of:

`Unable to create diagnostics file` *or*
`Unable to create statistics file`

Possible cause: Either:

The program does not have write access in the directory where the diagnostics or statistics files are to be created.

or

There are too many files open already.

Action: Check the access permissions to the current directory (using **`ls -l`**, and change them if necessary (using **`chmod`** *n*); close files as soon as they are no longer needed.

`Unable to open process log file`

Possible cause: Either:

The program does not have write access in the directory where the log file is to be created.

or:

There are too many files open already.

Action: Check the access permissions to the current directory (using **`ls -l`**) and change them if necessary (using **`chmod`**); close files as soon as they are no longer needed.

Appendix C

System error messages

System error messages are preceded by one of two initial messages, either:

```
DAP System Error:
```

`or:`

```
System Error:
```

C.1 DAP system error messages

If any messages preceded by **`DAP System Error:`** occurr, try re-running the program. If the error persists contact your AMT representative.

C.2 System error messages

If you get an error message preceded by **`System Error:`**, unless you get one of the error messages detailed below, you should re-run your program. If the error persists contact your AMT representative.

Either of:

`Attempted access to segment 0 by loader`, *or*
`DAP areas fragmented incorrectly when writing process log file`

Possible cause: Some sort of system error.

Action: Try re-running the program. If it fails again, contact your AMT representative.

`DOF file record structure invalid at offset` *n*

Possible cause: The DOF file has been changed since consolidation.

Action: Relink.

`Exec system call failed`

Possible cause: There are too many processes runing on your Sun.

Action: Quit any unnecessary windows, stop any unnecessary processes, and try again!

Either of:

Failure reading plog records from loader, *or*
Failure to open DOF file in loader

Possible cause: The DOF file has been deleted (or moved) during the program execution, or is corrupt.

Action: Restore the DOF file.

Failure to open system message file

Possible cause: The system message file **/usr/lib/dap/dap_msg_lib** does not exist or is not readable by the program.

Action: Check the permissions and reset to give everyone read access (set permissions with **chmod 1** *system-message-filename*. If the file does not exist contact your AMT representative.

Any of:

Failure when performing link driver ioctl, *or*
Failure when reading DOF file, *or*
Failure when reading common area records, *or*
Failure when reading diagnostic records, *or*
Failure when reading entry point or common area records, *or*
Failure when reading file, *or*
Failure when reading from array store, *or*
Failure when reading from support, *or*
Failure when reading line records, *or*
Failure when reading load reply from DAP, *or*
Failure when reading message from DAP, *or*
Failure when reading message from host, *or*
Failure when reading name attribute records, *or*
Failure when reading name records, *or*
Failure when reading process log file, *or*
Failure when reading section detail records, *or*
Failure when reading section table records, *or*
Failure when reading use of name records, *or*
Failure when writing down pipe, *or*
Failure when writing file, *or*
Failure when writing load abandon message to DAP, *or*
Failure when writing load complete message to DAP, *or*
Failure when writing load request to DAP, *or*
Failure when writing message to host, *or*
Failure when writing message to the DAP *or*
Failure when writing process log file, *or*
Failure when writing process log file data, *or*
Failure when writing process information to host, *or*
Failure when writing to support, *or*
Fork system call failed

Possible cause: The host computer has run out of virtual memory, or an internal error has occured.

Action: Reduce the number of processes and delete as many windows as possible.

`FORTRAN-PLUS trace request from APAL code section`

Possible cause: An internal error has occured.

Action: Try re-running the program. If it fails again, contact your AMT representative.

`Invalid fixup format in DOF file`

Possible cause: The DOF file has changed while the program is running.

Action: Check that no other processes (for example, linking) are still active when you are running the DAP program.

Any of:

`Invalid trace request SVC` *hex* **`received`**, *or*
`Loader internal message number` *n* **`out of range`**, *or*
`Loader message number` *n* **`from the DAP out of range`**

Possible cause: An internal error has occured.

Action: Try re-running the program. If it fails again, contact your AMT representative.

Either of:

`Malloc system call failed`, *or*
`Pipe system call failed`

Possible cause: The host computer has run out of virtual memory, or an internal error has occured.

Action: Reduce the number of processes and delete as many windows as possible.

`Segment 0 received by loader in load reply`, *or*
`Trace request instruction is not in a code section`

Possible cause: An internal error has occured.

Action: Try re-running the program. If it fails again, contact your AMT representative.

Any of:

`Transfer failure when executing dapsen or daprec`, *or*
`Transfer failure when reading SCA`, *or*
`Transfer failure when reading UPCA`, *or*
`Transfer failure when reading array store data`, *or*
`Transfer failure when reading file`, *or*
`Transfer failure when reading trace data block`, *or*

`Transfer failure when writing DAP program`, *or*
`Transfer failure when writing file`

Possible cause: The DAP has been disconnected during operation.

Action: Contact your system manager.

`Unable to restart program due to fatal error`

Possible cause: An attempt to restart the DAP program has occurred after an error, but restarting is impossible.

You might get this message if you specify the **`c`** option (continue) for the **`-e`** flag in **`dapopt`** .

Action: Correct your program; the error message displayed immediately before this '**`Unable`** ...' message should give you a clue to the cause of the original problem.

Either of:

`Unassigned error message`, *or*
`Unexpected EOF from DAP`

Possible cause: The DAP is turned off or has been disconnected during operation.

Action: Contact your system manager.

`Unrecognised CALL instruction` *hex* **`received`**

Possible cause: An internal error has occured, or the program has become corrupt.

Action: Try re-running the program. If it fails again, contact your AMT representative.

Either of:

`Unrecognised dapopt record read`, *or*
`Unexpected EOF when reading DOF options`

Possible cause: The DOF file has changed while the program is running.

Action: Check that no other processes (for example, linking) are still active when you are running the DAP program.

Any of:

`Unexpected message from dapsupport in Breakpoint Edit Mode`, *or*
`Unrecognised message received from support`, *or*
`Unrecognised or incomprehensible message received by support from host`

Possible cause: An internal error has occured, or the program has become corrupt.

Action: Try re-running the program. If it fails again, contact your AMT representative.

Appendix D

Messages from psam and dapdb

psam and dapdb give out three classes of messages: error messages, internal error messages, and warning messages.

D.1 Error messages

Error messages are preceded by:

```
Error:
```

The messages are:

`Address not specified`

Possible cause:	You have used the **array** command, but have not specified any address.
Action:	Specify the address whose contents you want to print (for more details, see **array**, on page 80.

`Attempt to select non-existent stack frame`

Possible cause:	You have used one of the stack navigation commands **up** or **down**, when the current procedure was at the top or bottom of the stack respectively.
Action:	Use the **backtrack** command to check the position of the current procedure on the stack.

`Cannot open process log file`

Possible cause:	The program cannot create the dumpfile *DOF-file-name*.**dr** or **dapcore**.
Action:	Change the permissions on the directory if necessary (using **chmod** *nnn log-file-name).*

`Column specified out of range`

Possible cause:	In the **array** command, you specified the *columns* parameter incorrectly (for more details, see **array**, on page 80).
Action:	Make sure that any column limits you specify are in the range 0 to (*ES*–1) , where *ES* is the DAP edge-size.

Count too large

Possible cause: You specified a *count* field for the **array** command that was too large.

Action: Reduce the size of the *count* field.

Data section or area name not recognised

Possible cause: The name you gave to the **array** command for a data section or area is unknown.

Action: Check the spelling of the data section name, and the consolidation map (use the **-m** option to **dapf** or **dapa**).

File does not contain requested dump

Possible cause: The dump number you gave to the **select** command is greater than the largest dump in the dump file.

Action: Select a different dump or possibly a different file.

File is not a process log file

Possible cause: The file you specified in the **core** command or as a parameter to **dapdb** is not a process log file.

Action: Check the spelling of the filename you gave.

First row or column larger than last row or column

Possible cause: A column limit you specified for the *firstcol* or *lastcol* parameter for the vertical format option of the the **array** command is out of range (for further details, see **array**, on page 80.

Action: Specify a correct limit. Note: the first column specified must be less than or equal to the last column specified.

Illegal character

Possible cause: There is an illegal character in the **array** command.

Action: Retype the complete **array** command line.

Illegal modifier

Possible cause: The syntax for a modifier is incorrect.

Action: The syntax is (**m***n*), where *n* is in the range 1 to 7; that is, is one of **(m1)** to **(m7)**.

`Incompatible APAL trace parameters`

Possible cause: The parameters you specifed to the **`array`** command are invalid.

Action: Look at the entry for the **`array`** command, on page 80.

`Integer or real precision specified is too large`

Possible cause: You specifed incorrectly the *size* field to the **`array`** command.

Action: Specify the field correctly. Note: the *size* field has to be between 1 and 64 for integers, and either 24, 32, 40, 48, 56 or 64 for reals.

`Item repeat count too large`

Possible cause: You specifed too large a *****count* field in the **`array`** command.

Action: Reduce the size of the *****count* field.

`Item repeat count too small`

Possible cause: You specified a negative *****count* field in the **`array`** command.

Action: Make the *****count* field greater than 0.

`Missing modifier`

Possible cause: When you specified a modifier to the *address* field in the **`array`** command, you did not specify a modifier register.

Action: Repeat the **`array`** command, and either remove the `(m`*n*`)` altogether, or insert a correct modifier register.

`Missing start-bit`

Possible cause: You specified an **`array`** command with rowpack format, but you specified a **/***start-bit* field without specifying a value for *start bit.*

Action: Insert a value after the `/`.

`Missing count`

Possible cause: You specified an **`array`** command with a *****count* field, but did not specify a value for *count.*

Action: Insert a value after the `*`.

`Missing offset`

Possible cause: You have specified an offset to the **address** field in the **array** command, but either you gave no offset value, or you gave an incorrect value.

Action: Insert a suitable value for the offset (for more details, see **array**, on page 80.

`Name missing from command`

Possible cause: You have omitted an obligatory parameter in a call to a command (for example, you issued the the **macro** command, but gave no *name* after it).

Action: Check the command syntax (for more details, see section 4.7, on page 80.)

`No FORTRAN-PLUS procedure selected`

Possible cause: You have issued the **print** command, but a FORTRAN-PLUS procedure has not been selected.

Action: Use the **backtrack** command to check what the selected procedure is, then select the desired FORTRAN-PLUS routine with the **procedure** command or with the stack navigation commands (**up**, **down**, **top**)

`No active procedure selected`

Possible cause: You have issued a stack navigation command (**up**, **down**, **top**), but there is no current active procedure.

Action: Select an active procedure using the **procedure** command or one of the stack navigation commands (**up**, **down**, **top**).

`No dump selected`

Possible cause: The dump file is corrupt.

Action: Rerun the program and retake dumps as required.

`No process log file selected`

Possible cause: You have issued a command to **dapdb** before you have selected a dump file.

Action: Use the **core** command to select a file.

`Number missing from command`

Possible cause: You have specified the **select** command, but have not specified the dump number required.

Action: Always specify the dump number.

`Number not recognised`

Possible cause: You have used an illegal character where a number was expected (for example, **`select 5s`**).

Action: Numbers have to be positive integers (hex numbers start 0x, octal numbers start 0, all others are decimal).

`Precision for data format used illegal`

Possible cause: You have specified the *size* field incorrectly to the **`array`** command.

Action: The *size* field has to be between 1 and 64 for integers; one of 24, 32, 40, 48, 56 or 64 for reals; 8 to *ES* (and a multiple of 8) for characters; and 1 to *ES* for hex and bit format.

`Procedure name not recognised`

Possible cause: You have give an unknown procedure name as a parameter to the **`procedure`** command – or have entered a non-command name on the command line when a non-FORTRAN-PLUS procedure is selected.

Action: Use the **`backtrack`** command to find out the active procedures, or look at the map you get with the **`-m3`** option to **`dapf`** or **`dapa`** to find the names of procedures in the program.

`Row specified out of range`

Possible cause: You specified incorrect row limit(s) (*firstrow* or *lastrow*) for the vertical format for the **`array`** command (for more details, see array, on page 80).

Action: Specify row limits in the range 0 to (*ES*-1).

Either of:

`Separator needed,` *or*
`Startbit only available in rowpack format`

Possible cause: You specified */ start-bit* for the **`array`** command, but did not specify rowpack format.

Action: Select rowpack format (using **`r`**) – for more details, see **`array`**, on page 80.

`Stack top pointer (M6) invalid,` *or*

Possible cause: The stack has been corrupted – which is normally because you tried to take a dump before control has passed to the DAP, or because in APAL you had used non-standard DAP-calling conventions.

Action: Make sure you only take dumps after a call to **`DAPENT`** or (if you are using APAL) use the standard calling conventions (for more details, see [2], *DAP Series: APAL Language*, section 9.2)

`Startbit too large, last bit exceeds row`

Possible cause: You have specified an **`array`** command in which the sum of *start-bit* and the *size* of the data item you want to print is greater than DAP edge-size.

Action: Reduce *start-bit* or the size of the data item.

Either of:

`System stack base pointer invalid,` *or*
`System stack top pointer (LNB) invalid`

Possible cause: The stack has been corrupted – which is normally because you tried to take a dump before control has passed to the DAP, or because in APAL you had used non-standard DAP-calling conventions.

Action: Make sure you only take dumps after a call to **`DAPENT`** or (if you are using APAL) use the standard calling conventions (for more details, see [2], *DAP Series: APAL Language*, section 9.2).

`Too many address components`

Possible cause: You specified *address* for the **`array`** command incorrectly.

Action: Specify the **`array`** command correctly (for more details, see **`array`**, on page 80.

D.2 Internal error messages

Internal error messages are those messages which indicate that an inconsistency has been detected in the debugger's tables. All such messages are preceded by

`Internal error:`

Any of:

`Illegal type specified,` *or*
`Illegal size specified,` *or*
`Message number out of range`

Possible cause: You should not see any of these messages.

Action: Contact your AMT representative.

D.3 Warning messages

Some problems are not considered, by **`psam`** or **`dapdb`**, to be classed as errors – in which case a warning message is sent to standard output (usually your host screen). Warning messages are preceded by

`Warning:`

The messages are:

`File opened but has no dumps`

Possible cause: The process log file is corrupt.

Action: Rerun the program and retake dumps as required.

Either of:

`Ignoring modifier` *number*, *or*
`Ignoring row/word offset`

Possible cause: You have specified a non-plane-aligned *address* for the **`array`** command with vertical format.

Action: **`psam`** and **`dapdb`** will take the address as that of the start of the plane containing the given address. If you want to specify an offset from the start of this plane, you should use the *firstrow* and *firstcol* directives (for more details, see **`array`**, on page 80).

`Ignoring word offset`

Possible cause: You have specified a non-row-aligned *address* for the **`array`** command with rowpack format.

Action: **`psam`** and **`dapdb`** will take the address as that of the start of the row containing the given address. If you want to specify an offset from the start of this row, you should use the */start_bit* directive (for more details, see **`array`**, on page 80).

`No information for these variables`

Possible causes: Either:

> You have used the **`print`** command to request information either on a variable that is in a FORTRAN-PLUS procedure that is not active.

or:

> When you compiled your program, you did not specify that diagnostic information on variables was to be collected – you specified a parameter of less than 2 to the **`-D`** flag to **`dapf`** (the default is to assume a value of 2 for **`-D`**). For more details, see section 2.7, on page 32.

or:

> You have entered a command or name that **`psam`** or **`dapdb`** does not recognise.

Action: Check that the variable is in an active routine, and has been assigned value(s).

Check that when you compiled your program, you either accepted the default for diagnostic information collection, or that you specified -2 for the `-D` flag.

Check (with **`psam`**'s **`history`** command) that you issued a valid **`psam`** command.

`No user-defined error interrupt masks`

Possible cause: You have issued the **masks** command, but have not yet defined any error interrupt masks.

Action: Check your DAP program.

`Not a FORTRAN-PLUS procedure`

Possible cause: Your current procedure is an APAL code section, and you have issued a **psam** command – such as **print** – that is only valid when a FORTRAN-PLUS procedure is current.

Action: Check which procedure is current (using the **backtrack** command), selecting a FORTRAN-PLUS procedure if appropriate (using the **procedure**, **up**, **down** or **top** command).

`Not an active procedure - no stack frame selected`

Possible cause: You have issued a command which is only valid when an active procedure is selected (for example, **print** or **attributes**), but the current procedure is not an active one.

Action: Select an active procedure (using the **backtrack** command to find out which procedures are active).

`Stack frame associated with non-Code address`

Possible cause: The stack is corrupt – which is normally because you have tried to take a dump of your DAP state (with a <CONTROL-\>) before you called **dapent**, or because you have used non-standard DAP calling conventions in your APAL program.

Action: Make sure that you only take dumps after you have called **dapent**, and (if you are writing APAL) you use the standard calling convention (for more details, see [3], *DAP Series: APAL Language* section 9.2).

`Using M6 as stack top pointer (LNB invalid)`

Possible cause: The standard copy of the LNB address is invalid (the standard copy is usually a copy of register M6). The problem usually happens when you write or use APAL programs which do not follow the standard calling conventions.

Action: Use the standard entry and exit macros (**'prologue** and **'epilogue**) for subroutine calls (see [3], *DAP Series : APAL Language*, section 9.2).

Appendix E

DAP interface routines

E.1 C language routines

NAMES **dapcon**, **dapent**, **daprec**, **daprel**, **dapsen** – DAP interface subroutines and functions

SYNOPSES

```
int dapcon (dap-prog-name)
char *dap-prog-name;

void dapent (dap-entry-name)
char *dap-entry-name;

void daprec (dap-common-name, word-aligned-data-area, size)
char *dap-common-name;
int *word-aligned-data-area;
int size;

void daprel ( )

void dapsen (dap-common-name, word-aligned-data-area, size)
char *dap-common-name;
int *word-aligned-data-area;
int size;
```

DESCRIPTION

dapcon requests access to the DAP, waits until permission is given and then the DOF file whose name is pointed to by *dap-prog-name* is either loaded into the DAP or into the simulator. **dapcon** returns an integer value indicating success or failure (see **Diagnostics** below). The DAP connection is released by calling the interface routine **daprel**.

dapent transfers control from the host C program to the DAP and returns control when a **return** statement in the DAP entry subroutine is executed. *dap-entry-name* is a pointer to the name of the DAP entry point to which execution is to be transferred.

daprec reads data from the DAP. **dapsen** sends data to the DAP and waits for the data to be transferred. They both take the same parameters. *dap-common-name* is a pointer to the name of a DAP **common** block from or to which data is to be transferred. *word-aligned-data-area* is a pointer to the start location of the block into which the data is to be read (**daprec**) or from which it is to be sent (**dapsen**).

WARNING The data area must be word-aligned. If necessary, it should be unioned to a word-aligned variable or placed in a word-aligned structure.

size is an integer variable specifying the number of DAP words (32 bits) to be transferred.

daprel has no parameters and releases control of the DAP, making it available to other users or programs. If a host program which is not connected to a DAP calls **daprel** no action is taken.

DIAGNOSTICS

Return codes from **dapcon** are as follows:

Result returned by **dapcon**	*Meaning of result*
0	Success
1	Unable to open DOF file
2	Unable to read DOF file
3	Not a DOF file
4	Unable to open channel to DAP
5	DAP load failed

NOTES You link these subroutines in a C host program by using the **-l** flag with **dap** as its argument.

For example, to compile program **hostprog.c** and call the output file **hostprog**, you could issue:

```
cc  -o  hostprog  hostprog.c  -ldap
```

E.2 FORTRAN language routines

NAMES **dapcon**, **dapent**, **daprec**, **daprel**, **dapsen** -- DAP interface functions and subroutines

SYNOPSES **integer function dapcon(***dap-prog-name***)**
character *(*) *dap-prog-name*

subroutine dapent(*dap-entry-name***)**
character *(*) *dap-entry-name*

subroutine dapsen(*dap-common-name, word-aligned-data-area, size***)**
character *(*) *dap-common-name*
integer (*) *word-aligned-data-area*
integer *size*

subroutine daprec(*dap-common-name, word-aligned-data-area, size***)**
character *(*) *dap-com-name*
integer (*) *word-aligned-data-area*
integer *size*

subroutine daprel

DESCRIPTION

dapcon requests access to the DAP, waits until permission is given and then the DOF file *dap-prog-name* is either loaded into the DAP or into the simulator. **dapcon** returns an integer value indicating success or failure (see **Diagnostics** below). The DAP connection is released by calling the interface routine **daprel**.

dapent transfers control from the host FORTRAN program to the DAP and returns control when a **return** instruction in the DAP entry subroutine is executed. *dap-entry-name* is the name of the DAP entry point to which execution is to be transferred.

daprec reads data from the DAP. **dapsen** sends data to the DAP and waits for the data to be transferred. They both take the same parameters. *dap-common-name* is the name of a DAP **common** block from or to which data is to be transferred. *word-aligned-data-area* is the name of the host data area into which data is to be received (**daprec**) or from which data is to be sent (**dapsen**).

size is an integer variable specifying the number of DAP words (32 bits) to be transferred.

WARNING The data area must be word-aligned. If necessary, it should be **equivalenced** to a word-aligned variable or placed at the start of a **common** block.

DIAGNOSTICS

Return codes from **dapcon** are as follows:

Result returned by **dapcon**	*Meaning of result*
0	Success
1	Unable to open DOF file
2	Unable to read DOF file
3	Not a DOF file
4	Unable to open channel to hardware DAP
5	DAP load failed

NOTES

You link these subroutines in a FORTRAN host program by using the **-l** flag with **dap** as its argument.

For example, to compile program **myprog.f** and call the output file **myprog**, you could issue:

```
f77 -o myprog myprog.f -ldap
```

Index

psam commands (such as **array**) are listed in this index under their own names, not under **psam**.

An entry such as **#include** is listed under its first alphabetic character, **i** in this case. All other non-alphabetic entries to the index are grouped together under the **!** heading immediately below this introduction.

AMT
Reading, UK

Reader comment form

Any comments you care to make, whether reporting bugs in the manual or making more general comment, about this or any AMT publications will help us improve their quality and usefulness. To report bugs, if you have the time, the ideal way from our point of view is to send us a photo-copy of the relevant page, with the bug marked on it. If you are in the UK, please use our FREEPOST address to send us the copy.

If you also can spare the time to fill in the mini-questionnaire below that would be doubly useful to us. To send us this form, please fold it as indicated, and post it – postage is pre-paid for the UK.

Comments

Title of publication: **DAP Series: Program Development under UNIX** (man003.04) / other – please specify:

. . . .

My name and job title: . . . .

My department: . . . .

My company: . . . .

My company address: . . . .

. . . .

My telephone number – country: . . . . number: . . . .

I used the publication:

- □ As an introduction to the subject
- □ To teach myself
- □ To teach others
- □ As a reference manual
- □ Other – please specify

I found the contents:

	True	Partly true	Not true
Helpful	□	□	□
Accurate	□	□	□
Written clearly	□	□	□
Well illustrated	□	□	□
Well indexed	□	□	□
Other – please specify	□	□	□

. . . .

Thank you for your help.

23 May 89

Second fold

First fold

Third fold

Third fold

No postage needed for posting in the UK.
If posting outside UK, please stick stamps to normal value.

Publications Manager
Active Memory Technology Ltd
FREEPOST (RG 1436)
Reading
Berkshire RG6 1BR
United Kingdom

Fourth fold

Fourth fold

Tuck into third fold

Second fold

First fold