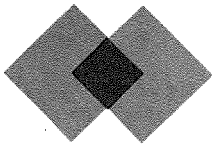


DAP Series

Engineering Test Software

DAPET

(man008.04)

AMT endeavours 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 AMT equipment should be read and understood by the operator before the equipment is used. To ensure that AMT equipment functions without risk to safety or health, such procedures should be strictly observed by the operator.

The development of AMT 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 AMT.

Specifications and statements as to performance in this document are AMT 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 licence save in so far as they are incorporated into a contract or licence by express reference. Issue of this document does not entitle the recipient to access to or use of the products described, and such access or use may be subject to separate contracts or licences.

Technical publication man008.04

(AMT filenames: asb\pubs\det\ed04\...)

First edition	January 1988
Second edition	February 1988
Third edition	27 June 1988
Fourth edition	23 May 1989

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

Publications Manager
Active Memory Technology Ltd
65 Sultons Park Avenue
Reading
Berks, RG6 1AZ, UK

Tel: { 0 / +44 } 734 661 111

Publications Manager
Active Memory Technology Inc
16802 Aston St Suite 103
Irvine, California, 92714,
USA

Tel: (+1) (714) 261 8901

Preface

dapet is the DAP engineers' test software – a suite of programs designed for users and support engineers.

dapet runs in the host system, and runs on both the types of host currently supported by AMT: that is, Sun workstations running under UNIX, and DEC VAX and MicroVAX workstations running under VAX/VMS. You use **dapet** in a similar way whichever host system you are using; the differences in usage are detailed in the relevant sections of this manual.

The manual is divided into two sections:

- Section I : tells you how to run the diagnostic test software supplied by AMT with your basic DAP-Series software. The diagnostic software lets you carry out a confidence check on your DAP, whether it is a DAP 500 or DAP 600 machine; it also lets you carry out fault diagnosis to board level
- Section II : contains more information for those users who build or maintain their own DAP systems and who need more facilities

If you have not used **dapet** before, then it is important that you read chapters 1 and 2 in section I first.

References

DAP Series: Engineering Service Guide	AMT	man008
DAP Series: System Management under UNIX	AMT	man019
DAP Series: System Management under VAX/VMS	AMT	man020

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 boxed, as well as 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 came 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
DAPET>
```

  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 the Shift key (either Shift key if there are two), and then press the 'Q' key

command syntax

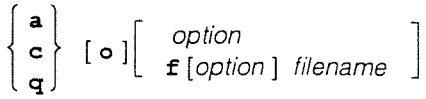
- The syntax for a command specifies optional and alternative sub-items in the command as:

  [ ] The item(s) enclosed in square brackets are optional. If included one and only one may be present

  { } One and only one of the items enclosed in braces must be specified

  . . . The item preceding the ellipsis may be repeated zero or more times; that is, the item may occur one or more times

For example, a command might be specified as:



Possible variations of the command include:

da

dao

da *option*

dao *option*

daf *filename*

daof *filename*

daf *option filename*

daof *option filename*

dc

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

Table of Contents

Section I

Using `dapet` to run test programs

Chapter 1

dapet general commands

1.1 Introduction

dapet is designed to check the operation of both DAP 500-series and DAP 600-series machines. It is easily accessible to all users, and meets a variety of needs. The confidence-checking and board-level diagnosis use of the test software, described in section I of this manual, is backed up by the in-depth testing facilities also available in **dapet** and described in section II of the manual.

This chapter describes how to enter the **dapet** environment, and how to keep a **dapet** session log; it also describes the help facilities that exist within **dapet**.

under UNIX and VAX/VMS – and upper and lower case

dapet runs almost identically on Sun and DEC VAX/VMS systems; any differences are noted in the relevant parts of the manual. One of the differences concerns the use of upper and lower case when you are typing commands or filenames. On VAX/VMS systems, you can name files and type commands in upper and lower case (or a mixture of both!), as VMS is not case-sensitive. By convention upper case is used on VAX/VMS, and this convention is followed in this manual for any commands you would only use in the VAX/VMS environment.

However, UNIX systems *are* case sensitive, so you must get the case right – at least for UNIX commands and filenames. **dapet** is not case-sensitive, so once you have entered **dapet** on either system, you can use upper or lower case for commands. For consistency, lower case is used for all non-specifically-VAX/VMS commands throughout this manual.

1.2 Establishing the test environment

under UNIX

If you are running under UNIX, you should type:

```
dapet
```

WARNING!!

UNIX does not stop you from running **dapet**, if **dapboot** is still running. If you try, you will get the message:

```
Warning, DAPBOOT is still active
```

and are likely to corrupt any user programs that are still running. Be warned!

For details of how to stop **dapboot**, see AMT's *DAP Series: System Management under UNIX*.

under VAX/VMS

If you are running under VMS, you should not use **dapet** if the **DAPMONITOR** program is running, as **dapet** will corrupt any user program in the DAP.

So before you try to run **dapet**, stop **DAPMONITOR**, by typing the command:

```
$ DAPBOOT STOP
```

To run **dapet**, first you should introduce **dapet** as a foreign command, by typing:

```
$ DAPET := $DAPETn
```

where *n* is 5 (if you want run **dapet** on a DAP 500) or 6 (if you want to run **dapet** on a DAP 600). You can then invoke **dapet** by typing:

```
$ DAPET
```

Note that you need operator (**OPER**) and **WO RLD** privileges to run **dapet**.

if the **DAPMONITOR** program is running, you will see the warning mesage:

```
Warning: DAPMONITOR is running
```

If in addition one or more DAP process are running, you will see a second message on your VAX screen:

```
Warning: Other processes using the DAP
```

You should stop **DAPMONITOR** with the **DAPBOOT STOP** command, as described above – having closed down any active DAP processes.

1.3 Welcoming message and screen format

Once **dapet** is invoked the system will output a message on the host VDU screen telling you the version of **dapet** that is installed on your system, and the DAP code store and array memory size. A typical message might be:

```
Version 3.1 with SCSI HCU link
MCU code size 512 KBytes, array size 4 MBytes
DAPET>
```

The prompt DAPET> tells you that the utility is ready to accept a command.

As noted above, you can issue any **dapet** command by typing upper or lower case letters, or any combination of the

two. If you are running under UNIX, don't forget that when you come to specify filenames, you need to specify the case of filenames exactly.

1.4 Help facilities

When you are in **dapet** you can get help on a command, by typing at the DAPET> prompt:

help *name*

where *name* is a **dapet** command name, or an allied DAP topic. If you type **help** on its own, you will get outline information on the entire set of **dapet** commands, and on the allied topics. You can then select any of those topic or command names to get further information.

Help information is fairly limited, but it can give you some idea of the scope of a command and its typical syntax and usage.

exiting from help – only under VMS

Only if you are running under VMS: when you want to exit from **help**, press the **RETURN** key, or type <CONTROL-Z>.

1.5 Saving a session log

You can keep a record on the host file system of your subsequent activity within a **dapet** session by issuing the command:

save *filename*

saving the log under UNIX

If you are running under UNIX, *filename* can be the name either of:

- An existing file to be overwritten
- A new file to be created in the current directory

saving the log under VMS

Under VMS, *filename* can be the name either of:

- An existing file, in which case VMS will create a new version of that file
- A new file to be created in the current directory. If you do not supply a file-type, VMS will add **.LOG** to *filename*

under UNIX and VMS

Once you have issued the **save** command, all commands, prompts and messages that are displayed on the host screen will be logged to the file, until you terminate the log by using the command:

saveoff

Note that any log file you have opened but not closed will be closed automatically by the system once you leave **dapet**, so saving the record of all your activity in the test session since you opened the file.

1.5.1 Quitting dapet

You can leave **dapet** whenever a DAPET> prompt appears on the host screen, by issuing either of the commands:

```
quit
```

or

```
q
```

quitting – under VMS only

Only if you are running under VMS: you can also quit **dapet** by typing <CONTROL-Z>.

Chapter 2

dapet basic user commands

2.1 Introduction

This chapter describes the syntax for the basic user commands in **dapet**; advanced user commands are described in chapter 3.

Once you have entered **dapet** you can run the tests either individually, or all in sequence automatically.

2.2 The test menu

The menu of tests available is displayed when you type:

menu

A typical menu is shown below:

```
Test no.    Testname    Test location   Test description
1           MCUTEST1    HOST           MCU host based test
2           MCUTEST2    MCU            MCU based MCU test part I
3           MCUTEST3    MCU            MCU based MCU test part II
4           MCUTEST4    MCU            MCU based MCU test part III
5           ARRAYTEST   MCU            Processor Element Array test
6           STORETEST   MCU            Array Memory test
7             DISTURB   MCU            Disturbance Test
```

2.3 Test organisation

Tests are subdivided internally into *subtests* and then further subdivided into *elements*. An element consists of a **dapet** software fragment which can be used to test a DAP instruction or a machine hardware feature. Subtests are groups of elements which, taken together, test a complete range of DAP features. Tests are groups of subtests which, taken together, test all the features of the selected part of the DAP.

Generally, the tests are designed to be run sequentially, with the lowest number test checking the most basic function of the machine. If you use the automatic test run feature, the basic features of the machine are checked for correct working, before the more advanced features are tested.

The first test – **MCUTEST1** – is 'host based'; it carries out some basic hardware tests on the DAP by accessing control

locations in the DAP from the host. All the other tests are 'MCU based'; they work by running programs in the DAP, and checking the output against the expected output.

Appendix B on page 59 describes all the tests, subtests and elements within **dapet**.

2.4 Running tests automatically

The command you issue to run all the tests in sequence is:

auto

All tests will run to completion with no further input from you, unless one of the tests fails. Successful completion of the test run is shown on the host screen as:

```
*** END OF AUTOMATIC CONFIDENCE RUN ***
```

If one of the tests fails, the run will be halted, and an error message will be output to the host screen, followed by the auto prompt:

```
AUTO
```

The system is now waiting for you to take some action. The options available are:

- To continue with the current test, by using the command **cont**
- To go on to the next test, by using the command **next**
- To quit the auto test sequence, by using the command **exit**. You can also use <CONTROL-C> to stop a running test immediately

Sections 2.8 and 2.9 in this chapter give you some guidance on the types of error that can occur, and what you can do to get tests on your DAP running again. If your testing has uncovered a problem with your DAP that you cannot resolve, chapter 4 on page 21 tells you how to contact AMT to seek more help.

2.5 Loading a single test

You can load a specific test by using the command:

lo { *test_name* / *test_num* }

where *test name* or *test num* identifies one of the tests on the test menu (see page 7).

For example:

```
lo 1
```

will load the test **MCUTEST1**;

```
lo MCUTEST1
```

will load the same test.

possible problem with `lo`

A point to note:

- The space between **`lo`** and the name or number is important. If you leave it out you will get an error message

Once you have issued the **`lo`** command, the system will output a message. If you had selected the **MCUTEST1** test, the message might be:

```
Initialising codestore ....
Loading MCU component "MCUTEST1"
Software version - 3.1.
No.of elements - 35.
```

2.6 Running a single test

Once you have selected and loaded a test, you can run it by typing:

```
go
```

You can specify the start and end points of a test exactly, using numeric arguments after the **go**. See page 17 (in chapter 3), or the on-line **help** information, for more details.

As each element in a test is run, a message is output on the host screen of the form:

```
Running subtest 2 element 8 -
```

If the element has passed, then the message is increased to:

```
Running subtest 2 element 8 - element passed
```

Individual tests should run to completion without stopping. However, should **dapet** detect a fault while it is running a test element, test execution will stop, and one or more error messages will be displayed.

When the whole test has finished, the DAPET> input prompt returns to the host screen.

2.7 Halting a test

You can halt a running test by typing <CONTROL-C>, which will halt test execution immediately. The DAPET> prompt will then appear, allowing you to enter another command.

If you want to restart test execution where you stopped it, or where the system stopped it because an error was detected, use the command **cont**.

2.8 Types of error

Error messages are displayed on the host screen by the system whenever a problem is detected. Three different types of errors can occur.

command entry errors

- *Command entry errors*

  dapet will only recognise commands with the correct syntax, and will not try to execute wrongly-spelt or non-existent commands. If you enter an incorrect command, the system will respond with an error message, and the DAPET> prompt will re-appear to allow you to re-enter the command. You can check the correct syntax in appendix A, the command reference section of this manual (page 51), or by using on-line **help**.

under UNIX

Since a usual command entry error is to mistype a command name, a typical error mesage might be, if you are running under UNIX:

```
Cannot open file "ato.DM"
```

You typed **ato** instead of **auto**, the system knew of no command **ato**, assumed it was a macro you had written, but couldn't find the file **ato.DM** containing the macro (see chapter 6 for more on macros).

under VMS

If you are running under VMS, macro file extensions are **.DM5** (for DAP 500) or **.DM6** (for DAP 6 00), so if you are using a DAP 500 and you type **ato** instead of **auto**, you would get the message:

```
Cannot open file "ATO.DM5"
```

command execution errors

- *Command execution errors*

  Command execution errors occur if **dapet** has not been able to execute a command. Entering **lo 27** would cause such an error; although nothing is syntactically wrong with the command, an error is reported since there are only 7 tests in the test menu (see page 7). An error message is output by the system, and the DAPET> prompt re-appears.

  The error message you would get if you entered **lo 27** would be:

```
There is no Test for number 27
```

test execution errors

- *Test execution errors*

  Test execution errors occur if a test element runs incorrectly, and show that there is something wrong with the DAP. Two types of error report can be generated here, FATAL and ERROR; they are described in detail below.

2.9 Test execution errors

FATAL ERRORs are caused by the DAP failing and stopping the test element functioning correctly. **dapet** detects this DAP malfunction, and outputs an appropriate message. The error message is terse, since the problem is bad enough to stop correct basic DAP function. A typical fatal error sequence is:

```
Running subtest 3 element 2 - Element terminated - FATAL ERROR

1. Element terminated OK, incorrect return.
-Suspect test element codestore write.
-Suspect MCU interrupt generation / HCU interrupt transfer.
```

ERRORs are caused by the test element detecting an error in the DAP function it is testing. The diagnostic error message output will give details of the fault that has been found. Usually the error message ends by stating which DAP board is suspect; if there is an array error you will usually also be told which processor element chip is suspect.

Certain errors in the DAP will cause a single test element to output more than one error message sequence; this multiple reporting usually only happens when you are running an array test.

A typical diagnostic sequence is shown on the next page.

```
Running subtest 16 element 2 - Element finished - ERROR

Message number = 217
ERROR IN TESTING MASTER-SLAVE ON THE PE STORE PINS
After setting a discrepancy between master and slave, there was
an unexpected status shown in the slave bank in the board where
the error was expected.

This occurred first when the unique 1 was in
column 24 row 0.

Board 0 showed an unexpected error status of #00000020
when the expected status was #00000022.

Bank A was the master, and bank A was reset to create the
discrepancy

There were a total of 2 errors of this type.

Suspect Array board(s) 0 and PE chip(s) 1.
```

The numbering of array boards and PE chips is explained in appendix C.

In many cases you will find it useful to have confirmation of which board is faulty by continuing the testing, and seeing what other faults are detected by the tests. You can continue with the testing after a non-fatal error has been reported, either by typing **cont**, or by setting the error trapping to **trap fatal** before you start running the tests – see page 18 (in chapter 3) for more details of **trap**.

ERROR reports often point to the suspected failing hardware element(s). See appendix C for details of the physical positions of the various array chips in the DAP, and of how to locate a suspect board from the information **dapet** gives .

If you get an error message suggesting that *all* PE chips are suspect, for example:

```
Suspect Array board(s) 0 and PE chip(s) All
```

then the fault is more likely to be on the array board than in an individual PE chip.

Similarly, if all array boards are reported as faulty:

```
Suspect Array board(s) All and PE chip(s) 3
```

the actual fault is most likely to be in the distribution of signals to the array boards, implicating perhaps an MCU board or the back plane.

If two array boards or chips are faulty then you will see the reference numbers of both failing items reported in the message. If more than two boards or chips are faulty, you will get a message like:

```
Suspect array board(s) MAP #AA00 and PE chip(s) 2
```

This example was from a DAP 600. The hex number (#AA00 in this case) is to be interpreted as a bit pattern (1010 1010 0000 0000 in this case) representing the failing boards, with a '1' representing a failing board, and the most significant bit representing board 0. In the example, boards 0, 2, 4 and 6 are suspect (or more likely, some signal that serves all these boards is suspect).

LEDs on the boards

Each of the circuit boards in the DAP has a green and a red LED mounted on it. During the diagnostic tests carried out when the DAP is switched on, the lights flash to indicate the progress of the tests. If the DAP passes these power-up tests, the red and green LEDs go out on all boards, except the green LED on the HCU board, which stays on. Under normal conditions, when user programs are being run, no red or green LEDs should come on, apart from the green HCU LED, which should stay on.

***all* green LEDs should be on**

When **dapet** starts running each element, the green LEDs on all boards are turned on. During certain elements – in particular the Array tests – if any boards are suspect, the relevant green LEDs are turned off. In the example at the top of the previous page the green LED on array board zero would be turned off.

The LEDs do not give any additional information to that presented on the screen; however, the suspect board(s) are highlighted by the absence of lit green LEDs, so physical identification of the boards is made easier.

***no* red LEDs should be on**

Control of the green LEDs is available to any priviledged software; control of the red LEDs is only available at the micro-code level. If you notice that any of the red LEDs come on, you will also probably get a FATAL ERROR message, as discussed at the start of this section.

Chapter 3

dapet advanced user commands

3.1 Introduction

This chapter describes the syntax for the advanced user commands in **dapet**. These commands will normally only be used by field engineers, but users who do their own maintenance may find them of value.

The chapter also describes the advanced features of **dapet**, which allow you a wider control over the calling of **dapet** and the running of tests.

The commands listed in this chapter change the way in which the DAP tests are run; any of the commands can be used whenever the command prompt DAPET> appears. Before you use any of the functions described in this chapter, you must first load a test by using the **lo** command (see page 8 for details).

3.2 Options/qualifiers available when dapet is called

3.2.1 Under UNIX

The options introduced in version 3.1 of **dapet** allow you to:

select comms line

- Switch to a 9600 baud serial line for DAP-host communication, instead of using the normal SCSI interface, if you suspect the interface to be faulty

specify which version of dapet to be loaded

- Specify which version of **dapet** is to be loaded. Versions of **dapet** for DAP 500 and DAP 600 are held by the host; if you issue a call to **dapet** the host loads the appropriate version, referring to a suitable MCU address location to find the edge size of the DAP. If the MCU has a serious fault the wrong version of **dapet** might be loaded – the welcoming message you see on the host screen once **dapet** is loaded tells you which version is active. This option gives a manual over-ride to the automatic selection of the **dapet** version

specify that codestore not to be initialised for every test

- Specify that the DAP codestore is not to be initialised every time a **dapet** test is loaded, to reduce the time needed to load a test when a serial line is to be used for DAP-host communication. Codestore is initialised from the host. The time to initialise the codestore using SCSI communication is a few seconds at most; with serial line communication, it is more than 7 minutes, even for a 512 Kbyte codestore

full call to dapet

The full specification for a call to **dapet** is now:

dapet [**-s**] [**-t** *dap_type*] [**-N**]

where:

-s specifies that a 9600 baud serial line is to be used for DAP-host communication. The default is that the SCSI interface is used. If -s is not specified, but the SCSI interface is not working, the host will try to use the serial line, if there is one

-t *dap-type* specifies which version of the **dapet** software is to be loaded, over-riding whatever would have been loaded by default; *dap-type* can take either of the values 5 or 6

-N specifies that the DAP codestore is not to be initialised before a **dapet** test is loaded; the default is that codestore is initialised before each test

use the options with care!

You should only use these options with care; you will seldom need to use any of them. Points to note are:

- If you specify **-N**, you may find that **dapet** reports FATAL errors incorrectly
- You should only use the **-t** option if the wrong version of **dapet** is announced in the **dapet** welcome message as having been loaded. If the wrong version *is* loaded, please let AMT know as soon as possible

WARNING —
using /SERIAL slows dapet greatly

Every time you load a **dapet** test, the host initialises the whole of DAP codestore, via the DAP-host communication link currently in use. If the parallel interface is used, this initialisation takes a few seconds at most. If a serial line is being used for DAP-host communication, the initialisation will take more than 7 minutes, even for a 512 Kbyte codestore.

3.2.2 Under VAX/VMS

You can now invoke **dapet** using one qualifier, which allows you to:

select comms line

- Switch to a 9600 baud serial line for DAP-host communication, instead of using the normal parallel interface, if you suspect the interface to be faulty

Set up the foreign command symbol **dapet** by typing at the **$** prompt:

$ **DAPET := $DAPET***n*

where *n* is 5 if you want to use **dapet** on a DAP 500, or 6 if you want to use it on a DAP 600.

You can then invoke **dapet** with the command **DAPET**. Its full specification is:

$ **DAPET [/SERIAL]**

where **/SERIAL** specifies that a 9600 baud serial line is to be used for DAP-host communications. The default is that the parallel interface is used.

Before you invoke **DAPET /SERIAL** you must specify the 9600 baud serial line you want to use – by defining the logical name **DAP_SERIAL** as the name of the serial device you want to use.

Hence, if you want to invoke **DAPET** using device **TTA2:** as a serial line, you would type:

```
$  DEFINE DAP_SERIAL TTA2:
$  DAPET /SERIAL
```

If you don't specify **/SERIAL**, the host will try to use the parallel interface, but if that interface is not working, the host will then try to use a serial line, if there is one.

You should only use the option with care; you will seldom need to use it.

WARNING — using /SERIAL slows dapet greatly

Every time you load a **dapet** test, the host initialises the whole of DAP codestore, via the DAP-host communication link currently in use. If the parallel interface is used, this initialisation takes a few seconds at most. If a serial line is being used for DAP-host communication, the initialisation will take more than 7 minutes, even for a 512 Kbyte codestore.

3.3 Defining the start and end-point of a test sequence

The command **go** defined in the last chapter, takes zero, one or more parameters. The various forms of **go** are:

go { *subtest_num* | *subtest_num element_num* | *start_subtest_num start element_num* *end_subtest_num end_element_num* }

where the specified subtest and element numbers exist in the test you have loaded.

go on its own starts the test sequence from the first element of the first subtest and works its way through all the subtests and elements to the last element in the last subtest in the test. Other forms of the command allow you to specify the start subtest and element, or both the start and end subtests and elements.

3.4 Running one test element

You can run just one test element, by issuing the command:

run *subtest_num element_num*

which runs only the subtest and element defined in the command. As in all commands concerning test execution,

before you run part of a test you must first have loaded the whole test (see **lo** on page 8).

If either the subtest or element you invoke does not exist, then the system would return an error message similar to:

```
subtest 3,element 66 not found.
Invalid arguments given for command "RUN"
```

3.5 Changing the effect of error detection

Normally when a fault is detected test execution stops and control is returned to you, which is what you usually want.

However there are times when you might find it useful for the testing to continue, perhaps when you are running a test where many elements fail. You might also want to run through the whole test, saving any error messages into a file you have set up with a **save** *filename* command (see page 5 for details).

The command you would use is :

trap *parameter*

where *parameter* can take one of four values:

parameters to trap

- **all** – the default option, active whenever you enter **dapet**. If **all** is active, any error stops the execution of the current test element, and returns control to you
- **fatal** – stops the testing only when a fatal error occurs. If this option is in force only serious problems will stop the testing
- **none** – no error stops test execution
- **?** – displays the current trap status, giving a message such as:

```
Trap option is : FATAL
```

Note that all error messages are displayed on the host screen, whichever option is in force.

3.6 Running a test sequence repeatedly

Sometimes you might need an element, a sub-test, or an entire test to be executed repeatedly, for example when you suspect that there is an intermittent fault in the DAP. To carry out this test repetition you would use the command **rep**. You can use zero, one or two parameters to the basic command:

rep { *subtest_num* / *subtest_num* *element_num* }

Execution stops only when you type <CONTROL-C>, or if an error specified by the **trap** command occurs. **rep** with no parameters runs the entire loaded test repeatedly. If you specify a *subtest_num*, the selected subtest is executed repeatedly. If you specify both parameters, the selected element is executed repeatedly.

3.7 Running the automatic test sequence

You can run the automatic test sequence repeatedly, by issuing the command:

autocycle

Execution will continue until you type <CONTROL-C>, or a **trap**-specified type of error occurs. You can also use the commands **next**, **exit** and **cont**; they act in the same way as when used with **auto** (see page 8 for details).

3.8 Changing the master PE bank

The DAP PE array has two identical processor banks, A and B, either of which you can set as the master bank, the other bank being the slave. Most of the array test elements exercise only the master bank, so unless you test the slave bank you may not detect a fault in it. **dapet** has a command which lets you specifiy which bank is to be master, letting you test both banks. The command is:

master *bank_name*

where *bank_name* may be either A or B. Bank A is the master when the test environment is entered.

You only need to test both banks when you are running a test involving the array; currently only test 5 on the menu (see page 7 for details of the menu). When you use the **auto** or **autocycle** commands (see pages 8 and 19 for details) array testing is done twice automatically, first with bank A, then with bank B.

Chapter 4

Contacting AMT

If the test software reveals a problem with the DAP that you cannot resolve, you may wish to contact AMT.

Your contact point with AMT will be your local AMT Service Desk. At the moment AMT Service Desks, and the areas they serve are:

United States of America:

Active Memory Technology Inc
16802 Aston St Suite 103
Irvine
California, 92714
USA

Telephone:

+1	714	261 8901	(for international callers)
	(714)	261 8901	(for callers in the US)
1	800	288 4268	(toll-free, outside California)

Contact:

Customer Services Manager

Europe and the rest of the world:

Active Memory Technology Ltd
65 Suttons Park Avenue
Reading
Berkshire RG6 1AZ
UK

Telephone:

+44	734	661 111	(for international callers)
0	734	661 111	(for callers in the UK)

Contact:

AMT Service Manager

When you contact AMT, the fault resolution process will be speeded up if you have the following information to hand:

- A completed DAP PROBLEM report (copies were supplied with your DAP; extra copies are available from your local AMT Service Desk)
- The name and exact location of your site
- The nature and urgency of your DAP problem
- A printout of the failing test run (see save on page 5)
- Any other information relevant to the problem

Section II

Using `dapet` to diagnose faults

Chapter 5

dapet engineer's commands

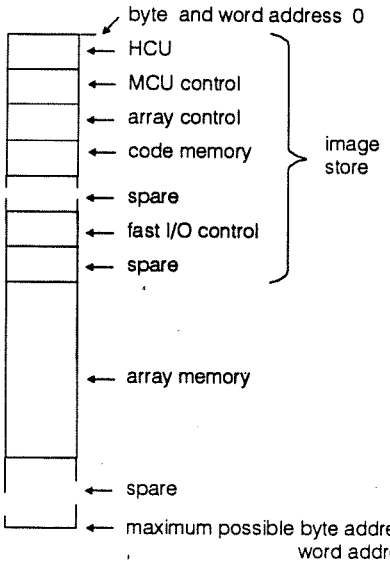
5.1 Introduction

This chapter describes the syntax of the engineer's commands in **dapet**. You are assumed to be familiar with the internal workings of the DAP, and with the terminology used to describe it. Appendix C to this manual gives the physical locations of the various array boards and array memory boards, and some of their chips, so that you can interpret the hardware diagnostic information **dapet** gives you.

The engineer's commands allow you to access and control every part of the DAP, including all control and data registers, and all of the system's memory. They are powerful commands, and should be used with care!

Although it is unlikely that you would damage the DAP hardware by using these commands, you could alter system configuration registers and stop the machine from working properly. If this happens, turn off the DAP, switch it back on again and execute **dapet**. The problem should now be cleared.

5.1.1 Image store and memory map



The DAP hardware includes many control and data locations (for the HCU, MCU, array, and fast IO) which you can access via **dapet** as you would normal memory. These locations, together with the code memory, are referred to here as *image store*, to distinguish them from the data memory associated with the DAP array – the array memory. The sketch in the margin gives you a rough not-to-scale picture of the memory map of a typical DAP. To use the industry-standard jargon, the DAP is a memory-mapped computer.

Word length in the DAP is 32 bits, and both word addresses and byte addresses can be used when accessing memory. Usually, when you are dealing with a particular part of the image store, say for MCU control, you specify addresses relative to the start of the MCU control part of memory, for which word addresses are used. Occasionally there is a need to use absolute addresses, which are given in bytes.

Access to image store is always to a single word (32 bits); access to array store is either to a complete row (one word on DAP 500, two words on DAP 600), or to a complete plane.

5.1.2 Groups of commands

Many of the commands in **dapet** fall into groups. For example, the commands **wax, wcx, wqx** and **wsx** are cases of the command **wn**x; **dia**, **dic**, **dif**, **dih** and **dim** are cases of the command **di***n*. The groups of commands are discussed in the sections that follow.

The meaning of the different possible values for *n* and **i***n* in the various commands are:

PE and array plane commands

- If *n* = **a**, the command is concerned with the 'A' plane
- If *n* = **c**, the command is concerned with the 'C' plane
- If *n* = **q**, the command is concerned with the 'Q' plane
- If *n* = **s**, the command is concerned with an array memory plane

image store commands

- If **i***n* = **ia**, the command is concerned with the array control part of image store
- If **i***n* = **ic**, the command is concerned with the code memory part of image store
- If **i***n* = **if**, the command is concerned with the fast I/O control part of image store
- If **i***n* = **ih**, the command is concerned with the HCU control and data part of image store
- If **i***n* = **im**, the command is concerned with the MCU control part of image store

5.1.3 Syntax conventions used

The following syntax conventions are used in this chapter (and elsewhere in the manual):

[ ]	The item(s) enclosed in square brackets are optional. If included one and only one may be present
{ }	One and only one of the items enclosed in braces must be specified
...	The item preceding the ellipsis may be repeated zero or more times; that is, the item may occur one or more times

5.1.4 Input of numeric values to commands

Unless otherwise stated in the sections that follow, any numeric values you input to **dapet** commands can only be expressed in hexadecimal notation. A hexadecimal prefix should not be used.

Some commands expect input values in a specific range; for example, some will expect as input a row address, or a value to be held in a row or word. In some cases the range they will accept depends on the DAP edge size. If you try to input a value greater than the relevant limit you will get an error message.

5.2 Image store write and display commands

Two groups of **dapet** commands allow you to write to or display the contents of various parts of image store.

To access part of the memory, you specify an address relative to the start of the section of memory you are concerned with. For example, if you want to access part of code memory, then the address you supply is relative to the start of code memory.

In the case of **dif** and **wif**, the addresses are relative to the image store address associated with the fast IO coupler specified by the most recent **sfio** command (see section 5.7 on page 34 for more details), or to coupler 0 if no such command was issued.

5.2.1 dia, dic, dif, dih, dim

dia, **dic**, **dif**, **dih**, and **dim** display the contents of image store. The usage is :

di { **a** **c** **f** **h** **m** } *addr* [*count*]

where *addr* is the word address of the start of the area to be displayed; *count* words are displayed, each with its corresponding address. If *count* is not specified only the word at location *addr* is displayed.

5.2.2 wia, wic, wif, wih, wim

wia, **wic**, **wif**, **wih**, and **wim** write to image store. The usage is:

wi { **a** **c** **f** **h** **m** } *addr val* [*count*]

where *addr* is the word address of the memory location to be written. If *count* is a valid hexadecimal number, then *val* will be written *count* times in consecutive word locations. If *count* is not a valid hex number, or is not given, then *val* is written at *addr* only.

5.3 PE and array write and display commands

In a DAP program you can both read from and write to any of the 3 processor element planes A, C and Q, although the DAP instruction set only caters for accessing a complete plane at a time.

WARNING!

PE plane access only indirect ...

For **dapet** to operate on such a plane, it copies across the whole of the relevant plane into a work area in array memory, then carries out your instructions on that copy. If you specified a write – either to a single row, or to the whole plane – the relevant part of the copy plane is changed, and the whole copy plane copied back to the processor element plane.

The work area is 3 planes deep, and by default sits at the high address end of the array memory, although the **setstore** command (see page 34 for details) allows you to move the work area to another location in the array memory.

DAP 600 edge register access also indirect ...

Similarly, on DAP 600 **dapet** can only access the edge register by copying it into a row in the work area, and operating on the relevant row in that plane.

... so you might see misleading effects

One point to note is that when you issue an access-PE plane command, you are not accessing the required plane, but a copy of it. A hardware fault in the work area in array memory, or in the DAP instruction decode circuitry could make you think that there was a hardware fault in a PE plane, or – only in a DAP 600 – in the edge register. Make sure that you have run the **dapet** auto test suite before you try to access PE planes or the DAP 600 edge register.

5.3.1 Use with DAP 500 and DAP 600

Although the array row lengths in DAP 500 and DAP 600 are different, the only difference between the row write to and read commands for the two ranges of machine is the maximum size of the arguments to the commands. The only groups of **dapet** commands affected by this difference are those covered in the two sections immediately following this one: sections 5.3.2 and 5.3.3

5.3.2 Row read – dax, dcx, dqx, dsx

dax, **dcx**, **dqx**, and **dsx** display processor element plane or array memory rows. The usage is :

d { **a** / **c** / **q** } **x** *row_num*

dsx *plane_addr row_num*

where *row_num* and *plane_addr* are appropriately-sized hexadecimal values, expressed without a hexadecimal prefix.

DAP 500

On DAP 500 *row_num* should lie between 0 and 1F.

DAP 600

On DAP 600 *row_num* should lie between 0 and 3F.

dsx displays row *row_num* of an array memory plane with plane address *plane_addr*.

5.3.3 Row write – wax, wcx, wqx, wsx

wax, **wcx**, **wqx**, and **wsx** write to processor element plane rows or array memory rows. The usage is as given at the top of the next page.

w { **a** / **c** / **q** } **x** *row_num val*

wsx *plane_addr row_num val*

where *row_num* and *val* are appropriately-sized hexadecimal values, expressed without a hexadecimal prefix.

DAP 500

On DAP 500 *row_num* should lie betwen 0 and 1F; *val* is stored as a 32-bit number.

DAP 600

On DAP 600 *row_num* should lie betwen 0 and 3F; *val* is stored as a 64-bit number.

wsx writes value *val* to row *row_num* of an array memory plane with plane address *plane_addr*.

5.3.4 Plane read – da, dc, dq, ds, dao, dco, dqo, dso, daf, dcf, dqf, dsf, daof, dcof, dqof, dsof

da, **dc**, **dq**, **ds**, **dao**, **dco**, **dqo**, **dso**, **daf**, **dcf**, **dqf**, **dsf**, **daof**, **dcof**, **dqof**, and **dsof** display processor element planes or array memory planes. The usage is:

d { **a** / **c** / **q** } [**o**] [*option* / **f** [*option*] *filename*]

ds [**o**] { *plane_addr* [*option*] / **f** *plane_addr* [*option*] f*ilename* }

where an *option* selection of '**b**' or '**B**' will display the plane contents in binary. **ds** will display the array memory plane at address *plane_addr*. The **o** option in the command name specifies the planes to be displayed orthogonally (that is, each line of data on the screen contains one column of data from the memory).

The **f** option specifies that the displayed plane's contents are also to be saved to a file in the current directory with name *filename*. Under VMS, unless you specify a file extension, **.LIS** will be added to your supplied *filename*. These files of the contents of planes can be used as input to the commands **waf** and so on, described in the next section, or as input to various **dapet** macro commands (see chapter 6 on **dapet** macros, starting on page 39).

5.3.5 Plane write – wa, wc, wq, ws, waf, wcf, wqf, wsf

wa, **wc**, **wq**, **ws**, **waf**, **wcf**, **wqf**, and **wsf** write to processor element planes or array store planes.

The usage is as given on the next page.

w { **a** / **c** / **q** } { *val* / **f** *filename* }

ws { *plane_addr val* / **f** *plane_addr filename* }

If option **f** is specified, then data is read from the file *filename* in the current directory, the first 'row' in the file to the first row in the plane, second row to second row, and so on until either all the rows have been written or the data is exhausted – a 'row' in a file is the same as a row in array memory: two words (each of 32-bits) for DAP 600, one word for DAP 500. The contents of any array memory row not written to are unaffected.

under VMS: default file type

Under VMS, if you do not supply a file type with *filename*, VMS will look for *filename* with type **.DP5** (if used on a DAP 500) or **.DP6** (on a DAP 600).

If **f** is not specified, *val* is written to every row of the specified plane. The information being written to the plane is also displayed on the host screen.

5.3.6 Inhibit read after write

Normally when you specify a write command – such as **wim** – **dapet** reads what it has just written, and sends the result back to your screen, allowing you visually to verify your write.

On occasions you may not want the read-after-write – for example, when you are synchronising an oscilloscope to the write, and don't want the display to be corrupted by the effects of the read.

The **disp** command allows you to switch off this read. The usage is:

disp { **0** / **1** }

where **0** specifies read-after-write (the default), **1** specifies inhibit read-after-write.

5.4 Use of absolute addresses

As suggested in the introduction to this chapter, the commands in sections 5.2 and 5.3 above all use addresses that are relative to the start of some area of DAP data or control memory. You will find that these commands are enough for most of your needs. Occasionally you may want to use absolute memory addresses, for which you would use commands **dm** and **wm**.

Different parts of the DAP hardware have different views of the DAP memory. The format for **dm** and **wm** is as seen by the HCU (that is, as seen by the host); byte addresses are used with **dm** and **wm**.

5.4.1 dm, wm

dm displays the contents of image store at specified absolute addresses. The normal use of the command is to display HCU or MCU memory locations. The usage is :

dm *addr* [*count*]

where the starting location is *addr*, specified in bytes. If no *count* is specified, then one word (4 bytes) is displayed. If a *count* is specified, then *count* words are displayed. *addr* should be a multiple of 4, so as to specify an address starting at a word boundary. The system will let you specify a non-multiple-of-4 *addr*, but the result may not be meaningful.

wm writes to image store at specified absolute addresses. The normal use of the command is to write to HCU or MCU memory locations. The usage is :

wm *addr val1* [*val2*] . . .

which writes the hex values *val1*, *val2*, and so on, to memory locations as words starting from hex *addr*, specified in bytes. Value *val1* is written to location *addr*; if *val2* is specified, it is written to location (*addr*+4), and so on, until the list of values is exhausted.

As with **dm**, *addr* should be a multiple of 4, so as to specify an address starting at a word boundary. The system will let you write to a non-multiple-of-4 *addr*, but the result may be unpredictable.

5.5 Display and alteration of system registers

There are many hardware registers in the DAP MCU, 25 of which are available to you to inspect and change. The names and hex numbers by which these 25 registers are known are:

M0	0
M1	1
.	.
.	.
M13	D
MP	F
CDATUM	10
CLIMIT	11
ADATUM	14
ALIMIT	15
DOSTART	16
DOCOUNT	17
DOITER	18
DOLEN	19
DOLOC	1A
PC	1B
JLOG	1E

Apart from the edge register **ME**, the registers missing from the list, for example registers 12, and 1C, are either reserved or as yet undefined.

edge register ME – different on DAP 500 and DAP 600

The edge register is in a special category, in that it is implemented differently on the DAP 500 and DAP 600.

on the '500

On the DAP 500, **ME** is located on one of the MCU boards, and can be accessed just like any of the 25 registers mentioned above, with a hex reference number of E:

ME E

The commands **regs, mreg** and **rreg** (see below for details) allow you to access **ME** in the same way as you access the 25 MCU registers listed above.

on the '600

On the DAP 600, **ME** is not located in the MCU and cannot be accessed in the same way as the MCU registers; commands **wedg** and **redg** (see below) give you access to **ME** on the DAP 600.

5.5.1 regs

You can inspect the contents of the 25 MCU registers (and **ME** on a DAP 500) with the **regs** command; it takes no arguments. Use **redg** to read a DAP 600 **ME**.

5.5.2 mreg

mreg allows you to alter one of the 25 MCU registers (and **ME** on a DAP 500). The usage is:

mreg { *reg_name* / **reg_num** } *val*

which writes the value *val* (interpreted as a hex number) to MCU register *reg_name*, where *reg_name* is one of the 25 names (26 on DAP 500) given above. *reg_num* may also be used to identify the selected register; the value you input is interpreted as a hex number. Use **wedg** to write to the DAP 600 **ME** (see below).

For **mreg** to have any predictable effect, the MCU should be in an idle state (see **mstop** on page 33) before you issue **mreg**; if the MCU is not idle when **mreg** is issued, then the effect on the selected register, and on the work the MCU is doing, is unpredictable.

5.5.3 rreg

rreg reads a specified MCU register. The usage is:

rreg { *reg_name* / *reg_num* }

which displays the value in the specified register. The usage of *reg_name* or *reg_num* is the same as in **mreg** (see section 5.5.2 above). To have a predictable effect, the MCU should be in an idle state (see **mstop** on the next page) before you issue **rreg**. Use **redg** to read a DAP 600 **ME**.

5.5.4 redg – DAP 600 only

redg lets you read the contents of the DAP 600 edge register, and displays its contents as a 16-digit hexadecimal number. The usage is:

```
redg
```

It takes no arguments. Note that the command uses indirect access to the register (see section 5.3 on page 28) and may not work correctly if there are faults with MCU instructions in group 2 (**raw**, **rax**, **rs**[o], **rw**[o], **rx**[o]) or group 6 (**sr**[n], **xr**[n], **wr**[n]).

5.5.5 wedg – DAP 600 only

wedg lets you write a 64-bit value to the DAP 600 edge register. The usage is:

wedg *val*

where *val* is a value expressed, without hexadecimal prefix, in up to 16 hexadecimal digits.

Note that the command uses indirect access to the register (see section 5.3 on page 28) and may not work correctly if there are faults with MCU instructions in group 2 (**raw**, **rax**, **rs**[o], **rw**[o], **rx**[o]) or group 6 (**sr**[n], **xr**[n], **wr**[n]).

5.5.6 asstate – DAP 600 only

The hardware Array Support unit, only present in the DAP 600, has various reflect and status registers, and **dapet** lets you inspect their contents, using the **asstate** command. The usage is:

```
asstate
```

It takes no arguments.

5.5.7 jlog

jlog displays the most recent 32 entries in the MCU jump log; the command takes no arguments.

5.5.8 pestate

In additon to the MCU and array support registers discussed above, you can inspect the contents of control registers associated with the array. **pestate** displays the MCU reflect registers, the PE master-slave error status registers, and the PE parity status registers; the command takes no arguments.

5.6 MCU control commands

5.6.1 mstop

mstop stops the MCU and puts it in an idle state; the command takes no arguments.

5.6.2 mstart

mstart starts the MCU; the command takes no arguments. Execution starts at the program counter value and machine state defined by the register PC (listed in section 5.5 above).

5.6.3 eint

eint sets the interrupt enable flag. Each test element checks the flag at the start of element execution; if the flag is set, all interrupts are enabled while the element is running. Interrupts are disabled by default, and are disabled at the end of each element. The command takes no arguments.

Some elements that test interrupt handling enable interrupts regardless of the state of this software interrupt enable flag.

5.6.4 dint

dint resets the interrupt enable flag to its default disabled state. The flag is read at the start of each test element execution; if the flag has been reset to zero no interrupts are enabled. The command takes no arguments.

When you enter **dapet** the interrupt enable flag is disabled. When you use the **auto** or **autocycle** commands, interrupts are enabled at the start and disabled at the end of the auto sequence. There is no automatic enabling of interrupts when you **load** and **run** tests individually; you can choose whether or not you use **eint**.

5.6.5 setstore

setstore sets processor element access planes. The usage is:

setstore *plane_addr*

In order to access PE planes for writing and reading, 3 array store planes are used as work areas. By default, these work areas are set to the last 3 planes in array store. See section 5.3 on page 28 for a discussion on the use of these planes.

The **setstore** command allows you to change the location of the first of these working planes to the address *plane_addr*.

5.7 Specifying the FIO coupler

the sfio command

The **sfio** command lets you specify which fast IO coupler is to be selected for testing. The default is that coupler 0 is selected.

dapet commands and programs that operate on the **sfio**-specified coupler include:

- **dif** and **wif** commands (see sections 5.2.1 and 5.2.2 on page 27)
- Test programs that exercise FIO couplers

The usage is:

sfio *fio_coupler_number*

where *fio_coupler_number* is a value in the range 0 to 3, or is the symbol **?**. If a value in the range 0 to 3 is used, the corresponding FIO coupler is selected; if **?** is used, the

command returns the number of the coupler already selected – whether by use of **sfio** or by default.

sfio causes bits 28 and 29 of the switch word (see section 5.9.3 on page 38) to be loaded with the specified coupler number, so individual **dapet** test elements can read the selected coupler number, and so use the appropriate image store address to access the coupler.

5.8 HCU task control commands

VRTX is the multi-tasking environment which runs on the HCU, and controls all the system and user-generated tasks run by the HCU. The commands discussed in this section give you some control over both user-generated and system tasks.

5.8.1 Loading VRTX tasks

You use the same command to load VRTX tasks as you do to load a **dapet** test:

lo *task_name*

loads the task in file *task_name*, where the specified file name includes, as usual, as much of its full path name as is needed to locate the file.

Under UNIX

Under UNIX the file must have an extension of **.hcu**, but you must not specify one in *task_name*.

Under VAX/VMS

Under VMS, you can specify a file-type in *task_name*. If you don't supply one, VMS will look for a file with file-type of **.HCU**.

loading VRTX tasks

Once loaded, the system assigns the task a VRTX task number, *task_id*. Although the task is then known to the HCU as both *task_id* and *task_name*, you get better control if you refer to it by its *task_name*; you also get more information from the report commands **vinq** and **vstat** if you use *task_name*. If you do use *task_id*, then the system will not carry out any error checking.

In all the commands that follow in this section only *task_name* is used, although *task_id* will evoke a similar response.

A task's initial status is DORMANT

5.8.2 vrun

vrun runs a user-generated dormant VRTX task. The usage is:

vrun *task_name*

where *task_name* is the name by which you identified the task to the HCU. If **vrun** is successful, the task's status is changed to RUNNING.

5.8.3 vsus

vsus suspends a running user-generated or system VRTX task. The usage is:

vsus *task_name*

where *task_name* is the name by which you identified the task to the HCU.

The status of the task must be RUNNING for the command to have any effect, in which case the task's status is changed to SUSPENDED.

5.8.4 vres

vres resumes a user-generated or system VRTX task after you have suspended it. The usage is:

vres *task_name*

where *task_name* is the name by which you identified the task to the HCU.

The status of the task must be SUSPENDED for the command to have any effect, in which case the task's status is changed to RUNNING .

5.8.5 vdel

vdel deletes a user-generated VRTX task from the system-held list. The usage is:

vdel *task_name*

where *task_name* is the name by which you identified the task to the HCU.

If the task's status is RUNNING or SUSPENDED, then the HCU task is deleted, and system knowledge of the task is removed.

If the specified task's status is DORMANT, the command has no effect.

5.8.6 vstat

vstat displays the current status of all the user-generated HCU tasks loaded in the current session. Also listed for each task are the task name, id and priority; and the load address, size and entry point. The usage is:

vstat

It takes no arguments. Note that **vstat** only knows about items loaded in the current session. Tables within **dapet** are searched, but the HCU is not interrogated by this command.

5.8.7 vinq

vinq requests the status of a user-generated or system task from the VRTX executive. The usage is:

vinq *task_name*

where *task_name* is the name by which you identified the task to the HCU.

vinq will give a more accurate and detailed status of a particular task than will **vstat**.

5.8.8 vpri

vpri changes the priority of a user-generated VRTX task. The usage is:

vpri *task_name priority*

where *task_name* is the name by which you identified the task to the HCU.

vpri will alter the priority of a VRTX task to *priority*, where *priority* is an integer in the range 0 to 255. Task priority increases as the value in *priority* decreases.

5.8.9 qclr

qclr clears one of the message queues controlled by VRTX. The usage is:

qclr *qid*

which clears the specified VRTX message queue *qid*.

5.9 Altering test element parameters

The design of **dapet** allows you to supply run-time parameters to a test element, although no elements that accept parameters have yet been released. The rest of this section documents the way you will be able to change the parameters for these yet-to-be-announced tests. More details of the way these parameters will be used will be released later.

5.9.1 setp

The user-modifiable tests refer to a 64-word internal parameter table for their detailed control. The default contents of the table is all-zeros, but you are able to use the command to insert your own values in 63 of the words in the table, or to inspect the contents of any of the words.

The usage is:

setp { *address value* | **?** }

setp *address value* writes *value* to the table at location *address*. Both *value* and *address* must be expressed in hexadecimal; *address* must be in the range 1 to 3f.

setp ? displays the contents of all 64 words in the table.

5.9.2 clrp

You are able to clear words 1 to 63 in the table by using the command:

clrp.

Word 0 is not affected.

5.9.3 sw

Word 0 in the parameter table is used as a switch word. You have access to the first 24 bits of that word, to allow you to pass information to the test, information which the test can use to control its execution.

The command:

sw

displays the contents of the switch word. Only bits 0 to 23 are displayed; the others are reserved for system use and are not accessible to you via the **sw** command.

altering switch word bits

The command:

on *bit position*

sets the specified bit position in the switch word to binary 1; the bit position must be expressed in decimal and must be in the range 0 to 23.

The command:

off *bit position*

sets the specified bit position in the switch word to binary 0; the bit position must be expressed in decimal and must be in the range 0 to 23.

5.10 Cycling a test element

The commands **autocycle** and **rep** described in section 3.7 (page 19) and section 3.6 (page 18) run the full test sequence or individual test elements repeatedly, and will output an error message whenever an error occurs.

If you are carrying out hardware fault analysis using an oscilloscope or logic analyser, you will sometimes need to run a specific element continuously, and to have all error messages suppressed. **dapet** provides this facility for any DAP-based test element with the command:

cycle *subtest_num element_num*

where *subtest_num* and *element_num* are the subtest and element number of the element you want to cycle.

Only a <CONTROL-C> will stop this command; if you try to cycle a host-based test, an error will be flagged.

Chapter 6

dapet macro commands

6.1 Introduction

This chapter describes the syntax for the macro commands in **dapet**.

dapet macros are much the same as macros in other command languages. You can write your own macros to carry out specialised tests, using commands which are described in this chapter, or using system commands, or using both; system-defined commands are those described in chapters 2, 3 and 5.

The commands described in this chapter allow you to load, store, compare, branch, jump, display, and carry out various logical functions.

Once you have written a macro, you store it away as an ordinary ASCII file. The macro can then be used in the same way as any other **dapet** command. A user-written macro can call other user-written macros.

The macro language is interpretive; detailed constraints on macros are given below.

6.2 The engineering test macro environment

Detailed features of **dapet** macros and the way they are used are:

- Macros are assumed to be in the current directory
- There are two ways of calling a macro:
  - At the DAPET> prompt, type the name of the macro, followed by the values of any macro parameters the macro might need. For example:

```
DAPET>  mymacroname [any_macro_parameters]
```

would call macro **mymacroname**, using any parameters that might be specified in the call

  - At the DAPET> prompt, type the command **ma**, followed by the name of the macro and any parameters the macro might need. For example:

```
DAPET>  ma mymacroname [any_macro_parameters]
```

The way in which you specify a macro name varies with the host operating system.

calling macros under VMS

Under VMS, **dapet** will accept a filename complete with file-type, but if you do not specify one, **.DM5** (for DAP 500) or **.DM6** (for DAP 600) is assumed by default.

under UNIX

Under UNIX, you must not specify any extension, as **.DM5** (for **DAP 500**) or **.DM6** (for DAP 600) is added after whatever you type. For example, if you are on a DAP 600 and you type **frieda.dm6**, **dapet** will look for file **frieda.dm6.DM6**, and will report an error if it cannot find it!

32-bit holding registers, h0 to h7

- 8 internal 32-bit holding registers (variables) are defined and are allocated the names **h0** to **h7**. **h0** is a special read-only register and always contains the result of the last row read from DAP array memory

8 holding planes, p0 to p7

- 8 internal holding planes are defined and are allocated the names **p0** to **p7**. Holding planes contain a copy of one plane of the array. On DAP 500 these holding planes are organised as 32 rows, each of one 32-bit word; on DAP 600 they are organised as 64 rows, each of two 32-bit words. You can load a holding plane from a file. You can also load a holding plane from an array memory plane, and vice versa, but cannot write to a file from a holding plane (except via an array memory plane). The only test you can specify on holding planes is an equality test. **p0** is a special read-only plane and always contains the result of the last read from a DAP array plane

labels

- For control purposes within a macro you can have up to 10 labels; they may be used in any part of the macro. Execution passed by a jump instruction will start on the line immediately after the label. See section 6.3.3 (page 45) and section 6.3.4 (page 45) for details of how labels are used.

  There is an extra label, **ERRLAB**; system control jumps to **ERRLAB** whenever an error occurs in 3 complex macro commands (**wrtsti**, **wrtstc**, and **mchk**), so if you use any of these 3 commands you should have some code starting at **ERRLAB** to handle any errors. If you are not using any of these commands, you can use **ERRLAB** as a general purpose label

loop variables

- There are 10 independent loop variables, which you can use in the normal way, to carry out a set of instructions several times. You can nest loops to the full depth of 10. See section 6.3.4 (page 45) for details of how loop variables are used

macro nesting

- You can nest macros to a depth of 5; if you try to nest them beyond 5, execution of all the macros will stop, and you will be returned to command level

- All variables (holding registers, holding planes, loop variables, and labels) are specific to the current level of macro
- You can use up to 3 parameters to a macro command; if they are used, holding registers **h1**, **h2** and **h3** are initialised at the beginning of the macro to the actual parameter values. You can use these parameters to specify how the macro is to run, or perhaps for how long it is to run

user-written macro names

- User-written macro names can be up to 12 characters long. Because each macro is stored in a file with the same name as the macro itself, your host operating system defines which alpha-numeric characters you can use in the name.

  You are advised not to give a macro the same name as a **dapet** command or macro command (listed in this chapter, and in appendix A starting on page 51). If you do use such a name, then you must use the **ma** command to call the macro (see section 6.2 on page 39)
- You can only have one command per line of macro text. Blank lines are ignored, as will lines beginning with a '!'. You can use these ! lines for comments – as in:

  ```
  ! this is is comment
  ```

An example of a typical macro is given at the end of this chapter.

6.3 User-accessible macro commands

The dapet macro command language uses symbolic names for the system macro commands; a typical command might be:

```
ld  p3  p6
```

which loads into holding plane **p3** the contents of holding plane **p6**.

All numeric values are assumed by the macro command interpreter to be hexadecimal, unless stated otherwise below.

6.3.1 Load and store commands

In the following description of the commands, h_n, h_{n1} and h_{n2} represent any of the holding registers **h1** to **h7**, and h_m, h_{m1} and h_{m2} represent any of the registers **h0** to **h7**; p_n represents any of the holding planes **p1** to **p7**, and p_m, p_{m1} and p_{m2} represent any of the planes **p0** to **p7**.

ld - load

$$\texttt{ld} \begin{Bmatrix} h_n\ h_m \\ h_n\ val \end{Bmatrix}$$

load into register h_n the contents of register h_m, or the value *val*.

ld $\left\{ \begin{matrix} p_n\ p_m \\ p_n\ file \end{matrix} \right\}$

load into plane p_n the contents of plane p_m, or the contents of file *file*.

filenames under VAX/VMS

Under VMS, the default file type for *file* is **.DP5** (on DAP 500), or **.DP6** (on DAP 600), although you can supply any file type you like.

filenames under UNIX

Under UNIX, you must not supply an extension to *file*, as UNIX will add **.dp5** (on DAP 500) or **.dp6** (on DAP 600) to whatever you specify in the command.

ldii $h_{n1}\ h_{n2}$

load into register h_{n2} from the part of image store pointed to by register h_{n1}.

ldic $h_{n1}\ h_{n2}$

load into register h_{n2} from the part of code memory pointed to by register h_{n1}.

ld and ldii/ldic specify their arguments differently

Note the order of arguments in both **ldii** and **ldic**: the receiving register is specified in the second argument; in ld the receiving register is specified first.

st – store

stii $\left\{ \begin{matrix} h_{n1}\ h_{n2} \\ h_{n1}\ val \end{matrix} \right\}$

store in the part of image store pointed to by register h_{n1} the contents of register h_{n2} or the value *val*.

stic $\left\{ \begin{matrix} h_{n1}\ h_{n2} \\ h_{n1}\ val \end{matrix} \right\}$

store in the part of code memory pointed to by register h_{n1} the contents of register h_{n2} or the value *val*.

clr – clear all registers/planes

clrh

clear all holding registers, that is, including register **h0**.

clrpl

clear all holding planes, that is, including plane **p0**.

stco – store 4 instructions

stco h_n *val1 val2 val3 val4*

write 4 instructions whose machine code values are *val1*, *val2*, *val3* and *val4* into DAP code memory, starting at the location pointed to by register h_n.

write-and-test commands

wrtsti h_m *val*

write value *val* to image store location pointed to by register h_m (using code similar to that used in **stii**), read that location (using code similar to that used in **ldii**), check if the result of the read is the same as the input to the write, if not jump to label ERRLAB.

wrtstc h_m *val*

write value *val* to code store location pointed to by register h_m (using code similar to that used in **stic**), read that location (using code similar to that used in **ldic**), check if the result of the read is the same as the input to the write, if not jump to label ERRLAB.

mset – set all MCU registers

mset

set into each mcu register (**m0** to **m13**, **mp**; **me** on DAP 500 only) an appropriate bit pattern. (You would use command **wedg** to set the DAP 600 edge register – see page 33).

The registers and the bit patterns in each of the registers after **mset** has been run are:

m0	A0A0A0A0
m1	A1A1A1A1
m2	A2A2A2A2
.	
.	
m13	ADADADAD
me	AEAEAEAE
mp	AFAFAFAF

on DAP 500 only

mchk – check a register

mchk $\left\{ \begin{matrix} h_m \\ val \end{matrix} \right\}$

check the bit pattern in the specified MCU register against the bit pattern that should be loaded into the register by the **mset** command; the register is specified by the contents of h_m or the value *val*; jump to label ERRLAB if the bit patterns are not the same.

You can use **mchk** to check the DAP 500 edge register; you would use command **redg** to read the DAP 600 edge register (see page 33).

6.3.2 Arithmetic and logical commands

add

add $\left\{ \begin{matrix} h_n \ h_m \\ h_n \ val \end{matrix} \right\}$

add the contents of register h_m or the value *val* to the contents of register h_n and put the result back in h_n.

sub

sub $\left\{ \begin{matrix} h_n \ h_m \\ h_n \ val \end{matrix} \right\}$

subtract the contents of register h_m or the value *val* from the contents of register h_n, and put the result back in h_n.

shl

shl $\left\{ \begin{matrix} h_n \ h_m \\ h_n \ val \end{matrix} \right\}$

planar shift left *s* times the contents of register h_n, where *s* is the contents of register h_m or the value *val*.

Note, **dapet** has no cyclic shift left command.

shr

shr $\left\{ \begin{matrix} h_n \ h_m \\ h_n \ val \end{matrix} \right\}$

planar shift right *s* times the contents of register h_n, where *s* is the contents of register h_m or the value *val*.

Note, **dapet** has no cyclic shift right command.

xor

xor $\left\{ \begin{matrix} h_n \ h_m \\ h_n \ val \end{matrix} \right\}$

EXCLUSIVE-OR the contents of register h_n with the contents of register h_m or the value *val*, and put the result back in h_n.

and

and $\left\{ \begin{matrix} h_n \ h_m \\ h_n \ val \end{matrix} \right\}$

AND the contents of register h_n with the contents of register h_m or the value *val*, and put the result back in h_n.

or

or $\left\{ \begin{matrix} h_n \ h_m \\ h_n \ val \end{matrix} \right\}$

OR the contents of register h_n with the contents of register h_m or the value *val*, and put the result back in h_n.

not

not h_n

carry out a one's complement on the contents of register h_n and put the result back in h_n.

6.3.3 Compare and jump commands

jmp

jmp *lab*

jump unconditionally to label *lab*.

jeq

jeq { *hm1 hm2 lab* / *hm1 val lab* }

jump to label *lab* if the contents of register h_{m1} are the same as the contents of register h_{m2} or the value *val*.

jpq

jpq *pm1 pm2 lab*

jump to label *lab* if the contents of plane p_{m1} are the same as the contents of plane p_{m2}.

jne

jne { *hm1 hm2 lab* / *hm1 val lab* }

jump to label *lab* if the contents of register h_{m1} are not the same as the contents of register h_{m2} or the value *val*.

jlt

jlt { *hm1 hm2 lab* / *hm1 val lab* }

jump to label *lab* if the contents of register h_{m1} are less than the contents of register h_{m2} or the value *val*.

jgt

jgt { *hm1 hm2 lab* / *hm1 val lab* }

jump to label *lab* if the contents of register h_{m1} are greater than the contents of register h_{m2} or the value *val*.

6.3.4 Looping and labels

label

label *lab*

insert label *lab*.

errlab

errlab

insert label ERRLAB.

setloop

setloop { *loop hm* / *loop val* }

set the $loop^{th}$ loop variable with the contents of register h_m, or with the value *val* (value expressed in decimal).

decjnz

decjnz loop lab

decrement the *loop*th loop variable by 1, and jump to label *lab* if the loop variable is not zero.

6.3.5 Display commands

dsh

dsh h_m

display the contents of register h_m.

dsp

dsp p_m

display the contents of plane p_m.

msg

msg *text*

print the text string *text*.

After *text* is dislayed, execution of the macro is suspended and the prompt:

```
MACRO>
```

is displayed. You can resume execution by typing **cont**, then pressing the **RETURN** key. Instead you can abandon macro execution and return control to **dapet**, by typing <CONTROL-C> (or <CONTROL-Z, on VAX/VMS hosts only).

text

text *text*

display the text string *text*.

6.3.6 Program control commands

pause

pause

pause in the execution of the macro.

When **pause** is obeyed, the macro execution is suspended and the prompt:

```
MACRO>
```

is displayed. You can resume execution of the macro by typing **cont**, then pressing the **RETURN** key. Instead you can abandon macro execution and return control to **dapet**, by typing <CONTROL-C> (or <CONTROL-Z>, on VAX/VMS hosts only).

exm

exm

exit the macro, and return to the next higher level of macro, or to the command level.

debug

debug

turn debug on (displays all macro statements before they are executed).

deboff

deboff

turn debug off.

mdir

under UNIX

mdir

under UNIX, display the names of all accessible macros in the current directory.

under VMS

mdir [*typ*]

under VMS, display the names of all accessible macros in the current default directory with file-type *typ* (default of **.DM5** on DAP 500, **.DM6** on DAP 600)

6.4 Macro commands available at command level

Between the execution of one macro and the next, the contents of the holding registers and planes are preserved, so you can examine them to check correct operation of the macro.

You can use the following macro commands to check operation:

dsh, **dsp**, **debug**, **deboff**, **mdir**

None of the other macro commands discussed in this chapter will be recognised in the **dapet** environment when the DAPET> prompt appears.

6.5 The auto macro

The **auto** command (discussed on page 8, and not itself a macro command) works by executing a special macro, which is stored in a text file.

location of file containing auto: under UNIX

Under UNIX, the **auto** macro for use on a DAP 500 is stored in file **AUTOSEQUENCE5**, for use on a DAP 600 in file **AUTOSEQUENCE6**; both flies are kept in the directory in which **dapet** resides.

under VMS

Under VMS, the **auto** macro is not stored in the same directory as **dapet**; for use on a DAP 500, it is stored in file

SYS$COMMON:[SYSMAINT.DAP]AUTOSEQUENCE5.; in file **SYS$COMMON:[SYSMAINT.DAP]AUTOSEQUENCE6.** for use on a DAP 600.

6.6 A typical dapet macro

The macro listed below is a typical macro; it carries out a similar function to subtest 5 element 1 in the host-based test MCUTEST1, validating the **NULL** instuction, and checking that the instruction is completely ignored.

```
! This test validates the NULL instruction,checking it is completely ignored
! h1 code store, h2 mcu control, h3 data access register

                ld          h1      00000000
                ld          h2      00000000
                ld          h3      00000001

! set up code store (null, interrupt, halt, halt )

                stco        h1      ff000000 fe000000 f6000000 f6000000

! set M0 - M15 with own address pattern

                mset

! set PC c0000000 and start MCU

                mreg        PC      c0000000

                mstart

! set up data access reg to access PC

                stii        h2      dd

! load PC into h4 from data access register

                ldii        h3      h4

! check PC ,error if it isnt the right value

                jne         h4      c0000003    2

! check Mregs - h4 is now the MCU register index

                ld          h4      0
                setloop     1       16
```

```
label 1

                mchk        h4
                add         h4      1
                decjnz      1       1


! if we get to here the test has passed OK

text Test passed.

                jmp         3

! error message section

errlab

text Test failed.

text The M register with the following index has been corrupted :

                dsh         h0
                jmp         3


label 2

text An invalid PC value was found after the test :

text The PC value expected = c0000003

text Actual PC value = :

                dsh         h4


label 3

                exm
```

Appendix A

dapet commands and macro commands

A.1 dapet commands

This section lists all the **dapet** commands and their possible parameters, and the pages in the manual where the commands are described.

commands for DAP 500 or DAP 600 only?

In general all dapet commands can be used on DAP 500 and DAP 600 systems, although there are a few that can only be used on one system; these are marked accordingly.

Where a command is described in more than one place in the manual, all places are listed here. The first place usually describes a simple use of the command, later places a more complex use.

The commands are:

asstate (on DAP 600 only) — display the contents of the registers in the Array Support Unit (page 33)

auto — load and run all the test programs in sequence (pages 8, 19)

autocycle — load and run repeatedly the sequence of all the test programs (pages 19, 19)

clrp — clear the whole of contents of the test parameter table to zeros (page 37)

cont — continue at the start of the next test element of a test or series of tests that have been stopped (pages 8, 10, 12, 19, 46)

cycle *subtest_num element_num* — cycle repeatedly through the specified MCU-based subtest element (page 38)

deboff — turn **debug** off (page 47)

debug — turn **debug** on (page 47)

di { **a** | **c** | **f** | **h** | **m** } *addr* [*count*] — display the contents of the image store, starting at the specified word address (page 27)

dint — reset the interrupt enable flag to its default state (page 34)

disp { **0** | **1** } — turn on (**disp 0**) or off (**disp 1**) read-after-writing when writing to the image store (page 30)

d { **a** | **c** | **q** } [**o**] [*option* | **f** [*option*] *filename*] — display, optionally in binary, the contents of the specified PE plane, saving the display to a file if required(page v)

ds[**o**] { *plane_addr* [*option*] | **f** *plane_addr* [*option*] *filename* } — display, optionally in binary, the contents of the specified array store, saving the display to a file if required (page 29)

d { **a** | **c** | **q** } **x** *row_num* — display the contents of the specified PE plane row (page 28)

dsx *plane_addr row_num* — display the contents of the specified array store row (page 28)

dm *addr* [*count*] — display the contents of the image store, starting at the specified absolute byte address (page 31)

eint — set the interrupt enable flag (page 34)

exit — exit from the **auto** test sequence (pages 8, 19)

go [*subtest* | *subtest_num element_num* | *start_subtest_num start_element_num* *end_subtest_num end_element_num*] — start execution of a test, subtest or element, or range of subtests and elements that is already loaded (pages 9, 17)

help *name* — offer on-line help on the specified command or topic (page 5)

jlog — display the MCU jump log (page 33)

lo { *test_name* | *test_num* | *task_name* } — load the specified test or VRTX task (pages 8, 35)

ma *macro_name* [*parameter_list*] — run the macro *macro_name*, using the given parameters (page 39)

master *bank_name* — set the processor master bank to the specified bank (page 19)

mreg { *reg_name* / *reg_num* } *val*	write *val* to a specified MCU register (page 32)
menu	list the tests available in dapet (page 7)
mstart	start (or re-start) the MCU (page 33)
mstop	stop the MCU, putting it into idle (page 33)
next	restart test execution at the next test (pages 8, 19)
off *bit_position*	reset the specified bit in the switch word associated with the test element parameter table, to binary 0 (page 38)
on *bit_position*	set the specified bit in the switch word associated with the test element parameter table, to binary 1 (page 38)
pestate	display the contents of the MCU reflect and parity status registers (page 33)
qclr *qid*	clear the specified VRTX message queue (page 37)
quit (or **q**)	quit **dapet** (page 6)
redg **(on DAP 600 only)**	display the contents of the edge register (page 33)
regs	display the contents of 25 (26 in a DAP 500) of the MCU registers (page 32)
rreg { *reg_name* / *reg_num* }	display the contents of the specified register (page 32)
rep [*subtest_num* / *subtest_num element_num*]	repeat indefinitely the whole of the already-loaded test, or selected subtest, or selected element (page 18)
run *subtest_num element_num*	run the selected element (page 17)

save *filename*	open a file in which to save the log of a **dapet** session (page 5) (under VMS, the file is created with default file-type **.LOG**, unless you specify a file-type; under UNIX, there is no extension to *file name* unless you specify it)
saveoff	close the already-opened file into which the **dapet** session log is being saved (page 5)
setp *addr val*	write a value to the specified address in the test element parameter table (page 37)
setp ?	display the contents of all 64 words in the test element parameter table (page 37)
setstore *addr*	change the starting location of the 3 array store planes used as work space from that of the last 3 planes in the array store (page 34)
sfio *fio_coupler_number*	select the specified FIO coupler for testing (page 34)
sfio ?	display the selected FIO coupler number (page 34)
sw	display the contents of the first 24 bits of the switch word associated with the test element parameter table (page 38)
trap { *all* / *fatal* / *none* / **?** }	specify which type of error should stop a **dapet** test sequence, or (for **?**) display the current **trap** status (page 18)
vdel *task_name*	delete the specified user-generated VRTX task from the system list (page 36)
vinq *task_name*	display the status of the specified active VRTX task (page 36)
vpri *task_name priority*	change the priority of the specified user-generated active VRTX task (page 37)
vres *task_name*	resume the specified user-generated or system VRTX task after it has been suspended (page 36)

vrun *task_name*	run the specified VRTX task (page 35)
vstat	display the current status of all user-generated HCU tasks loaded in the current **dapet** session (page 36)
vsus *task_name*	suspend the specified VRTX task (page 36)
w { **a** / **c** / **q** } { *val* / **f** *filename* }	write to the specified PE plane, optionally reading the data from a file (page 30)
ws { *plane_addr val* / **f** *plane_addr filename* }	write to the specified array memory plane, optionally reading the data from a file (page 30)
w { **a** / **c** / **q** } **x** *row_num val*	write to the specified PE plane row (page 28)
wedg *val* **(on DAP 600 only)**	write to the edge register (page 33)
wsx *plane_addr row_num val*	write to the specified array memory row (page 29)
wi { **a** / **c** / **f** / **h** / **m** } *addr val* [*count*]	write to image store, starting at the specified word address (page 27)
wm *addr val1* [*val2*] ...	write to image store, starting at the specified absolute byte address (page 31)

A.2 dapet macro commands

This section of the appendix lists in alphabetical order all the macro commands available in dapet, and discussed in section 6.3 starting on page 33. The information here repeats that in section 6.3, so no references to section 6.3 are given.

The macro commands are:

add { h_n h_m / h_n *val* }	add h_m or *val* to h_n and put result in h_n
and { h_n h_m / h_n *val* }	AND h_m or *val* with h_n and put the result in h_n

clrh	clear all holding registers
clrpl	clear all holding planes
deboff	turn **debug** off
debug	turn **debug** on
decjnz *loop lab*	decrement the *loop*th loop variable by 1, and jump to label *lab* if the loop variable is not zero
dsh h_m	display the contents of register h_m
dsp p_m	display the contents of plane p_m
errlab	insert label ERRLAB
exm	exit the macro, and return to the next higher level of macro if there is one, otherwise return to the command level
jeq { h_{m1} h_{m2} *lab* / h_{m1} *val lab* }	jump to label *lab* if the contents of register h_{m1} are the same as the contents of register h_{m2} or the value *val*
jgt { h_{m1} h_{m2} *lab* / h_{m1} *val lab* }	jump to label *lab* if the contents of register h_{m1} are greater than the contents of register h_{m2} or the value *val*
jlt { h_{m1} h_{m2} *lab* / h_{m1} *val lab* }	jump to label *lab* if the contents of register h_{m1} are less than the contents of register h_{m2} or the value *val*
jmp *lab*	jump to label *lab*
jne { h_{m1} h_{m2} *lab* / h_{m1} *val lab* }	jump to label *lab* if the contents of register h_{m1} are not equal to the contents of register h_{m2} or to the value *val*
jpq p_{m1} p_{m2} *lab*	jump to label *lab* if the contents of plane p_{m1} are the same as the contents of plane p_{m2}
label *lab*	insert label *lab*
ld { h_n h_m / h_n *val* }	load into register h_n the contents of register h_m or the value *val*
ld { p_n p_m / p_n *file* }	load into plane p_n the contents of plane p_m or the contents of file *file*
ldic h_{n1} h_{n2}	load into register h_{n2} from the part of MCU code memory pointed to by register h_{n1}
ldii h_{n1} h_{n2}	load into register h_{n2} from the part of DAP image store pointed to by register h_{n1}
mchk { *val* / h_m }	check the bit pattern in MCU register h_m or *val* against the bit pattern loaded into the register by command **mset**; jump to label ERRLAB if the bit patterns are not the same
mdir [*typ*] (under VAX/VMS)	display the names of all accessible macros on the current default directory with file-type *typ* (default **.DM5** on DAP 500, **.DM6** on DAP 600)
mdir (under UNIX)	display the names of all accessible macros on the current directory

mset set all MCU registers with appropriate bit patterns. The patterns are:

m0	A0A0A0A0
m1	A1A1A1A1
m2	A2A2A2A2
.	.
.	.
.	.
m13	ADADADAD
me	AEAEAEAE
mp	AFAFAFAF

(on DAP 500 only)

msg *text* print text string *text* and pause in the execution (type **cont** to continue)

not h_n carry out one's complement on the contents of register h_n and put the contents back in h_n

or $\left\{ \begin{matrix} h_n & h_m \\ h_n & val \end{matrix} \right\}$ OR contents of register h_n with the contents of register h_m or with the value *val*, and put the contents back in h_n

pause pause in the execution (type **cont** to continue)

setloop $\left\{ \begin{matrix} loop & h_m \\ loop & val \end{matrix} \right\}$ set the $loop^{th}$ variable with the contents of register h_m or the value *val* (expressed in decimal)

shl $\left\{ \begin{matrix} h_n & h_m \\ h_n & val \end{matrix} \right\}$ planar shift left s times the contents of register h_n, where s is the contents of register h_m or the value *val*

shr $\left\{ \begin{matrix} h_n & h_m \\ h_n & val \end{matrix} \right\}$ planar shift right s times the contents of register h_n, where s is the contents of register h_m or the value *val*

stco *hn val1 val2 val3 val4* store 4 instructions whose binary code values are *val1*, *val2*, *val3* and *val4* in DAP code memory, starting at location pointed to by register h_n

stic $\left\{ \begin{matrix} h_{n1} & h_{n2} \\ h_{n1} & val \end{matrix} \right\}$ store in a location in code memory pointed to by register h_{n1} the contents of register h_{n2} or the value *val*

stii $\left\{ \begin{matrix} h_{n1} & h_{n2} \\ h_{n1} & val \end{matrix} \right\}$ store in a location in image store pointed to by register h_{n1} the contents of register h_{n2} or the value *val*

sub $\left\{ \begin{matrix} h_n & h_m \\ h_n & val \end{matrix} \right\}$ subtract the contents of register h_m or the value *val* from the contents of register h_n, and put the result back in h_n

text *text* print the text string *text* and continue execution

wedg *val* write into the edge register (DAP 600)

wrtstc h_n *val* write using **stic**, read using **ldic**; check if the result of the read is the same as the input to the write, if not jump to label **ERRLAB** (see page 34 for more details)

wrtsti h_n *val*

write using **stii**, read using **ldii**; check if the result of the read is the same as the input to the write, if not jump to label **ERRLAB** (see page 34 for more details)

xor { h_n h_m } { h_n *val* }

EXCLUSIVE-OR the contents of register h_n with the contents of register h_m or the value *val*, and put the result back in h_n

Appendix C

Locating suspect components

When a **dapet** test element fails, it usually tells you which board is suspect – see page 11 for a typical diagnostic report. Because the DAP 500 and DAP 600 are quite different in physical layout, this appendix is split into two different parts, one part for each type of machine.

suspect PE chip or array board

If a PE chip is suspect, **dapet** identifies that chip and the array board on which the chip is fitted. If more than one PE chip is implicated, then the fault could be in any one of the suspect chips, or on the array board that contains the suspect chips.

As suggested earlier in this manual, if **dapet** reports a fault, it is useful to continue the whole suite of **dapet** tests, as you will often get confirmation of the fault in the output from other test elements. If **dapet** reports more than one array board as suspect, then the fault could be in any one of the identified boards, or if all array boards are implicated, the fault may well be in one of the MCU boards.

suspect memory location

If a memory location is suspect, then the absolute address of that location is given.

This appendix gives you the physical locations of the various array boards and their daughter memory boards, and tells you how to work out from the suspect memory location which memory board is suspect.

C.1 DAP 500

Figure C.1.1 on the next page shows you the location of the various boards on the machine's backplane. There are four identical array boards in a DAP 500, which on the machine are numbered 0, 1, 2 and 3, according to the array rows they service. Note though, as figure C.1.1 shows, the boards are not in consecutive physical locations on the backplane.

For more details of the physical layout of a DAP 500 and of its technical functioning see AMT's *DAP 500: Engineering Service Guide* (man008).

C.1.1 Layout of the DAP 500 backplane

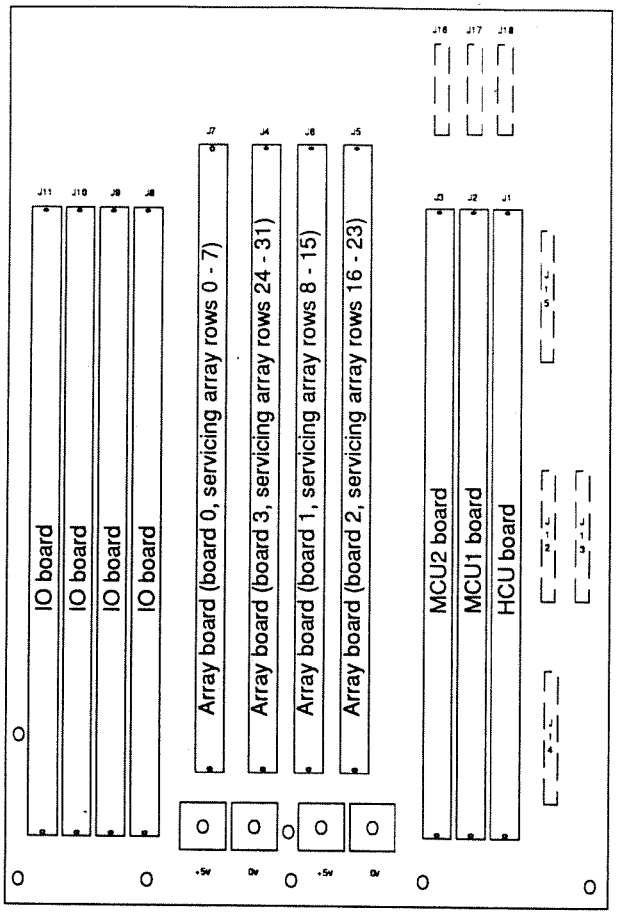


Figure C.1.1: *Layout of the IO, array, MCU and HCU boards on the backplane of a DAP 500, looking from the front of the card cage*

Note: **dapet** identifies faulty array boards as board 0, 1, 2, or 3; the correlation between these **dapet** numbers and the sockets into which the boards are plugged is as folows:

dapet *board number*	*Backplane socket number*	*Array rows on the board*
0	J7	0 – 7
1	J6	8 – 15
2	J5	16 – 23
3	J4	24 – 31

C.1.2 Layout of the PE chips on an array board

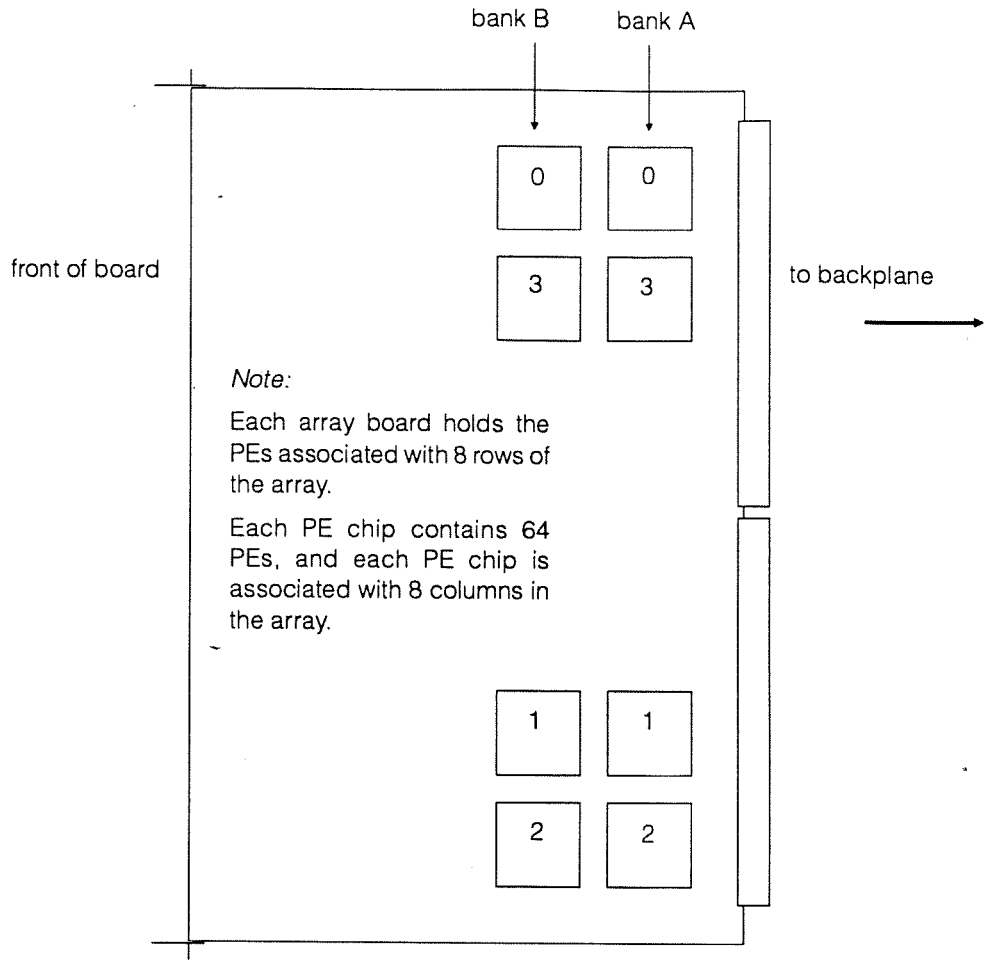


Figure C.1.2: Layout of PE chips on an array board, showing the **dapet** *reference numbers*

dapet reports a faulty PE chip as being 0, 1, 2 or 3; the correlation between these **dapet** numbers and the physical chip numbers (as recorded on the boards and in the engineering documentation) is as follows:

dapet PE chip number	*Columns in array*	*Physical PE chip reference:*	
		in bank A	*in bank B*
0	0 – 7	U4	U8
1	8 – 15	U2	U6
2	16 – 23	U3	U7
3	24 – 31	U1	U5

Generally, you should suspect a PE chip in the bank currently specified as master, except for a few test elements that perform their own bank selection, and which give explicit messages about the PE chip bank under test.

Note that bank A is used by default, both for **dapet** and for normal DAP programs.

C.1.3 Location of the array memory boards

Each array board has four memory boards piggy-backing on it, as shown below:

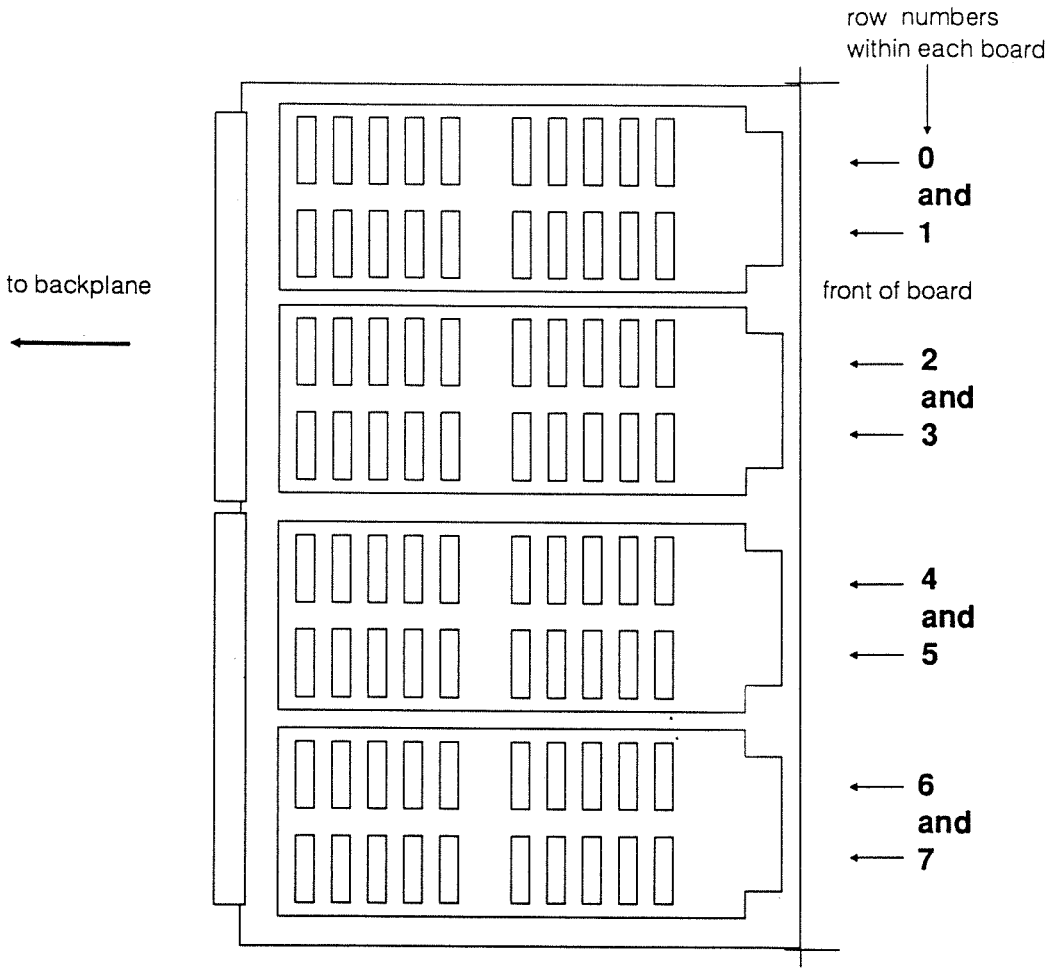


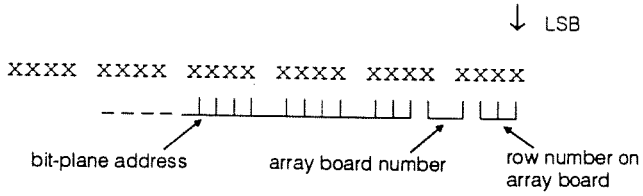
Figure C.1.3: Location of the array memory boards on the back of a DAP array board, showing the PE array row numbers associated with each memory board

Note: In some DAP machines the array memory boards are not fully populated; the layout of memory components on the memory boards may differ in detail from that shown above.

C.1.4 Calculating faulty memory board locations

If **dapet** suspects a memory chip is malfunctioning, it reports an error at an absolute memory address, given in words (not bytes). From that address you can work out which row of memory chips mounted on which array board is suspect, and hence which array memory board is suspect.

On DAP 500 the **dapet**-reported address is interpreted as follows:



The number of relevant bit-plane address bits depends on the size of your DAP's array memory. If you have 4 Mbytes of array memory, for example, then 15 bit-plane address bits are relevant.

The method of calculating the faulty board is detailed below, taking as an example a **dapet**-reported suspect memory location of F392A:

- From the address of the suspect memory location, extract the 5 least-significant bits (the array board and row numbers of the suspect location).
  Example: address – F392A, 5 least significant bits – 0 1010
- The top 2 bits of those 5 least significant bits give the array board on which the suspect memory chip resides.
  Example: address – F392A, suspect array board – 1
- The bottom 3 bits of those 5 least significant bits give the row on the array memory board holding the suspect memory chip.
  Example: address – F392A, row holding the suspect chip – 2

C.2 DAP 600

In DAP 600 half of the possible 24 boards are plugged in from the front of the cabinet, and the rest from the back. A single double-sided backplane in the centre of the cabinet is used for both sets of boards.

C.2.1 Board layout in a DAP 600

Figures C.2.1 and C.2.2 below gives you views of a DAP 600 card cage looking into the cabinet from the front and from the back.

In the front view (figure C.2.1) you can see, looking from left to right, array boards 8 to 15; the ASU, the two MCU, and the HCU

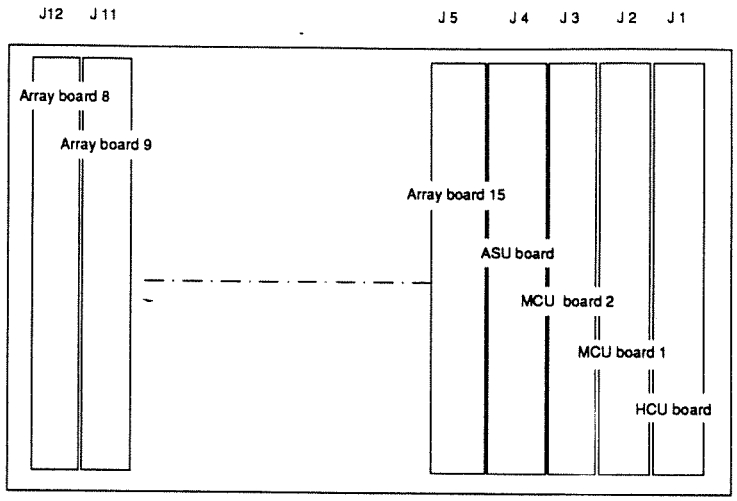


Figure C.2.1: A view from the front of the cabinet of a populated DAP 600 card cage, with the backplane slot numbers superimposed on the drawing for easy reference

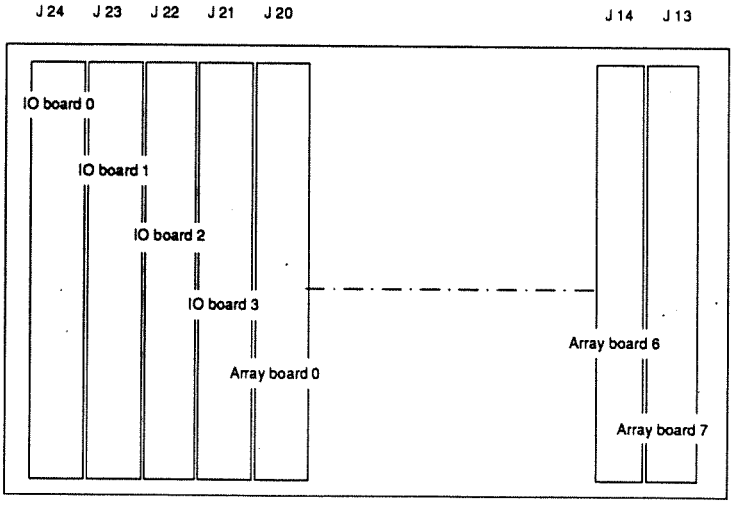


Figure C.2.2: A view from the back of the cabinet of a fully-populated DAP 600 card cage, with the backplane slot numbers superimposed on the drawing for easy reference

boards. The back view (figure C.2.2) is of a fully populated cardcage. On the left are four IO coupler boards (not all the board may be present in your DAP – depending on your DAP's configuration). Next to these board are array board 0, array board 1, and so on, with array board 7 on the right of the cage.

The backplane slot numbers (J1, J2, and so on) which are normally covered by the boards, are also shown on the figures for easy reference.

depopulated backplane

Figure C.2.3 below shows a view from the front of the cabinet of a fully de-populated backplane. The board slots shown as solid rectangles are for boards that plug in from the front (those shown in figure C.2.1); the dotted rectangles are for those boards that plug in from the back (those shown in figure C.2.2). If you look into a de-populated cabinet from the front, you can see the back of the connectors associated with the boards that plug in from the back – and if you look in from the back, you can see the back of the connectors for the front-plug-in boards.

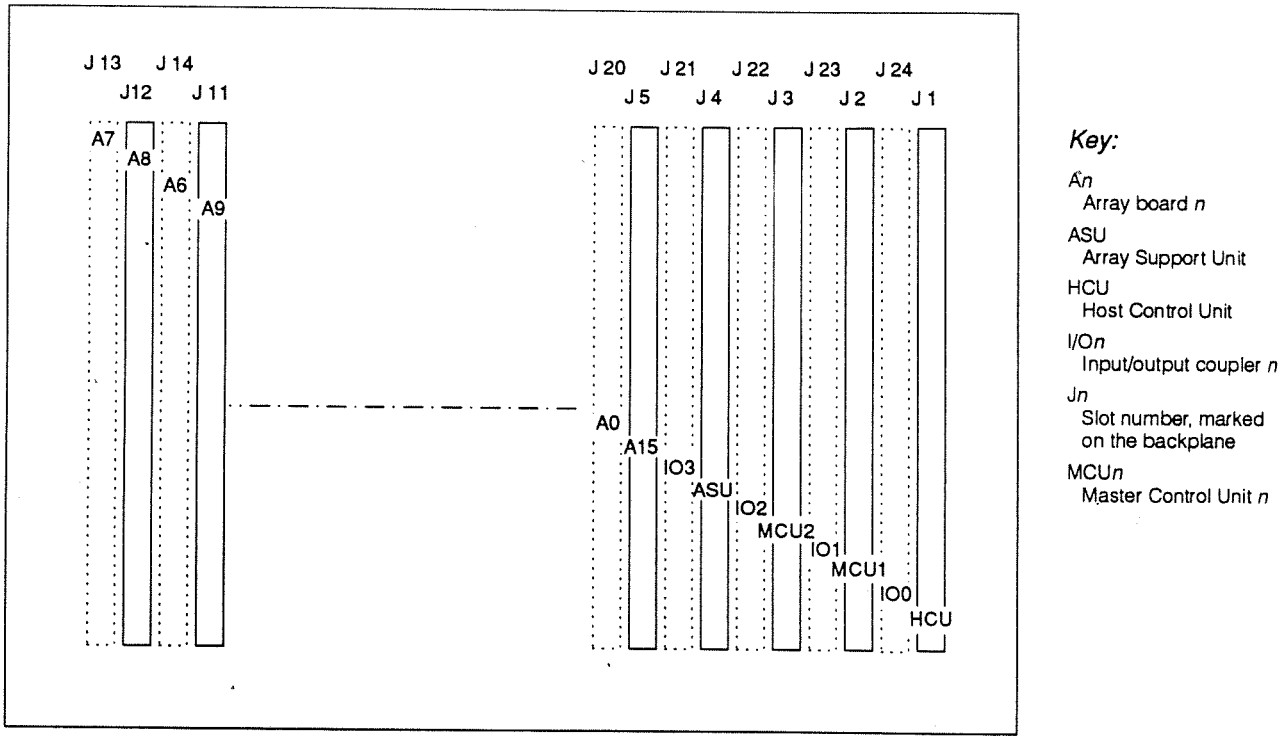


Figure C.2.3: View of an empty DAP 600 back plane from the front of the cabinet, showing the locations of the boards that plug in from the front (shown as full rectangles) and from the back (shown as dotted rectangles)

C.2.2 Array board numbering

Each array board deals with a section of the array of logical size 32 columns by 8 rows. The even-numbered boards deal with the more significant halves of 8 consecutive rows, and the odd numbered boards deal with the less significant halves. The table below details the relationship between array board numbers and the PE array rows and columns they 'provide'.

Board	0	Rows	0	–	7	Board	1	Row	0	– 7
		Columns	0	–	31			Columns	32	– 63
Board	2	Rows	8	–	15	Board	3	Rows	8	– 15
		Columns	0	–	31			Columns	32	– 63
Board	4	Rows	16	–	23	Board	5	Rows	16	– 23
		Columns	0	–	31			Columns	32	– 63
Board	6	Rows	24	–	31	Board	7	Rows	24	– 31
		Columns	0	–	31			Columns	32	– 63
Board	8	Rows	32	–	39	Board	9	Rows	32	– 39
		Columns	0	–	31			Columns	32	– 63
Board	10	Rows	40	–	47	Board	11	Rows	40	– 47
		Columns	0	–	31			Columns	32	– 63
Board	12	Rows	48	–	55	Board	13	Rows	48	– 55
		Columns	0	–	31			Columns	32	– 63
Board	14	Rows	56	–	63	Board	15	Rows	56	– 63
		Columns	0	–	31			Columns	32	– 63

Table C.2.1: Array board numbers and the PE array rows and columns they provide

C.2.3 Array board layout

The array boards in a DAP 600 are identical to those in a DAP 500; see figure C.1.2 for details of that layout.

All the general coments in section C.1.2 apply to the DAP 600 also, except that PE chip 0 (as reported by **dapet**) refers either to array columns 0 – 7 or 32 – 39, depending on whether it is in an even or odd numbered array board respectively. Corresponding comments apply to other PE chips.

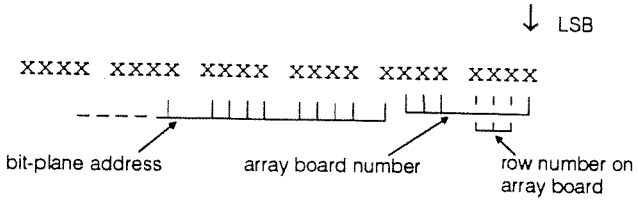
C.2.4 Memory board layout

Again, memory boards for the DAP 600 are identical to those for the DAP 500; see figure C.1.3 for details of that layout. A particular memory board provides the memory for the more significant halves of two rows, or the less significant halves of two rows, depending on whether the board is piggy-backed on an even or an odd numbered array board respectively.

C.2.5 Calculating faulty memory board locations

If **dapet** suspects a memory chip is malfunctioning, it reports an error at an absolute memory address, given in words (not bytes). From that address you can work out which row of memory chips mounted on which array board is suspect, and hence which array memory board is suspect.

On DAP 600 the **dapet**-reported address is interpreted as follows:



Note that the least significant bit of the array board number is separated from the board's other address bits.

The number of relevant bit-plane address bits depends on the size of your DAP's array memory. If you have 16 Mbytes of array memory, for example, then 15 bit-plane address bits are relevant.

The method of calculating the faulty board is detailed below, taking as an example a **dapet**-reported suspect memory location of D4216:

- From the address of the suspect memory location, extract the 7 least-significant bits (the array board and row number of the suspect location).
  Example: address – D4216, 7 least significant bits – 001 0110
- The top 3 bits and the bottom bit of those 7 least significant bits give the number of the array board on which the suspect memory chip resides.
  Example: address – D4216, suspect array board – 2
- The remaining 3 bits of those 7 least significant bits gives the row on the array memory boards holding the suspect memory chip.
  Example: address – D4216, row holding the suspect chip – 3

Index

This index lists all the commands and their associated parameters available to you in **dapet**; they are shown in **boldtype**. All non-alphabetic entries to the index are grouped together under the **!** heading immediately below this introduction.

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Reader comment form

AMT
Reading, UK

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: **Engineering Test Software** (man008.04) / 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