PDS

6502 Assembler & Monitor manual

Contents

1 The Programmers Development System 6502 Assembler

The assembler is invoked from the editor at any position in the source code by pressing the [ASSEM] function key. Assembly begins at the start of file 0 and will continue through the files until the end of file 7, or an END is reached. If an error occurs during assembly the assembler will stop, display an error message and move the current cursor position to the start of the line containing the error. Clear the error message by pressing any key, correct the error and reassemble again.

Each line is individually processed and assembled. The assembler is looking for a number of possibilities the line could contain.

Labels in PDS must start on the first character of the line. Labels can be of any length, containing any valid label characters for that particular assembler. The first character can be any valid label character, except the numeric digits 0 to 9. The valid label characters that are allowed in the 6502 assembler are:

! . 0-9 ? A-Z _ £

Note the assembler is not case sensitive, it converts everything into upper case. At the end of the label PDS will ignore one non-label character. This means that if your source code has come from an assembler that demands labels be followed with colons or hashes then PDS will ignore the extra characters. For example, PDS accepts all these syntaxes as label definitions for FRED:

```
FRED        NOP
FRED:       NOP
FRED#       NOP
```

This feature may cause a problems if you insert an extra character by mistake, as in the following examples:

```
FRED+       NOP
FRED-       NOP
FRED*       NOP
FRED;       NOP
```

Notice the semicolon in the fourth example does not cause the rest of the line to be ignored, it is treated as the label terminator. If you don't want to define a label on a line you must enter at least one space or tab. In between all the fields on a line, PDS will skip over any number of tabs and/or spaces. After a label PDS will expect to find a pseudo opcode or mnemonic. If you have an isolated label in a line, it is given the current value of program counter.

Comments may be included anywhere in the source code, they are preceded by semi colons. Note, never use a semi colon for a comment immediately after a label as it will read as the label terminator character. After a semi colon the rest of the line is ignored. At the very start of a line you may also use an asterisk to make the assembler ignore the line. For example:

```
; This is just a line of comment
**********************************
*           So is this           *
**********************************
            NOP                   ; This is a comment
```

2 Expression Evaluation

When a label is at the start of the line, it is set to the current value of the program counter (the program counter is the address where code is assembled to). You may have an number of labels, each having the same value. For example:

```
FRED
JOHN
BILL        NOP
```

FRED, JOHN, BILL all have the same value, the memory address of the NOP instruction.

Local labels are a major part of PDS, any label that starts with a pling sign (!) is considered a local label. You may use local labels in the same way you'd use normal labels, although every time you use a 'normal' label, all the local label definitions are cleared. We recommend that you use normal labels at the start of main subroutines, and local labels within the subroutine. This way you will never get re-defined label errors. Below is an example of local labels.

```
!1          NOP
            NOP
            JMP !1              ;(Jumps to the first !1)
FRED        NOP
            JMP !1              ;(Second)
            NOP
!1          BNE !1              ;(Second)
            NOP
BILL        NOP
            NOP
!1          JMP !1              ;(Third)
```

You should try and use local labels as much as possible. For short jumps (over a few bytes) it is not worth having to think up a valid label name. Local labels must be used in macros to prevent multiple definitions, but they will not affect other local labels with the same name, either in the main program, or in nested macros. Local labels also make it easy to copy large blocks of source code, because if all the labels are local, they need not be altered. Just ensure that you insert a normal label in front of the copied code.

If you wish to know the current address of the program counter (PC) in an expression then you may use '*', which will return the current PC. So the example below, BILL, JOE and JOHN are given the same value:

```
BILL        EQU *
JOE
JOHN        NOP
```

Numbers in PDS may be entered in HEX, DECIMAL, ASCII or BINARY.

To enter a number in decimal just enter the number as normal. PDS will allow you to enter numbers from -32768 to 65535, but remember that minus numbers are a duplication of the numbers 32768 to 65535. For example, if you entered -20/4 the expression would return 16379 instead of -5 as you might expect. This is because the -20 was treated as 65516 (the 2's complement value).

To enter a number in HEX, you may either enter it starting with an and sign (&) eg: &1234 or &ABCD or entering it starting with the more familiar dollar sign ($), eg: $1234 or $ABCD

To enter a number in BINARY, proceed the string of ones and zeros with a '%' eg: %1010 or %1001110

To enter ASCII values, just enclose them in double quotes, for example "a" returns $61, or "ab" returns $6261.

Expressions in PDS can contain labels, numbers and any number of brackets and operators. Expressions are evaluated from left to right using standard mathematical priorities, for example: 2+3*6 will return 20 not 30. The operators PDS

allows are as follows:

-	Negate, if a number or label, not an expression, is preceded by a minus sign then its value will be negated. If you wish to negate an expression then you should use, -1 * (expression).
+-*/	Will add, subtract, multiply and divide the two operands. There are no error checks on these operations, so you may have expressions such as 60000*5 or 60000+60000 which will return the result MOD 65535. Remember that division will round down, so 7/3 will be 2.
&	Will logical AND the two operands.
!	Will logical OR the two operands.
~	Will logical XOR the two operands.
=	If the two operands are equal, this will return one, else it will return zero. eg: 5=4 returns 0 and 23=23 returns 1.
<>	Is the reverse of the above.
>	If the expression on left hand side, is greater than the right hand side expression, then this will evaluate to one, otherwise zero is returned.
<	Is the reverse of the above.

If '>' or '<' is at the start of an expression, PDS will return either the low or high bytes only. For example >$1234 will return $34 and <$1234 will return $12. This is the default setting, the arrow pointing at the byte returned. You may reverse this setting in the configuration system, so that >$1234 would return $12. Note the greater or lesser than signs can only be used at the start of expressions, eg: >23*52+FRED. If you want to use high or low bytes within an expression you will have to surround the expression with brackets eg: 23+52*(<FRED).

The expressions are evaluated from left to right, except for operators with higher priorities. Operators are evaluated in the following order:

(and)	
< and >	- when used to return LSB and MSB.
* and /	
+ and -	
=, <>, >, and <	- when used as more than and less than.
&, ! and ~	

So 2+7&3 will evaluate 2+7 then AND with 3, returning 1. Or 7&3+2 would evaluate 3+2 then AND with 7, returning 5. Brackets can be used to override these priorities, as they have the highest priority.

A nice feature in PDS is the ability to check the range of expressions in the program code. If you want an expression to return a value only within a specific range, eg between 100 and 120, then simply insert the range directly after the expression, enclosed in square brackets. For example:

```
FRED*2+3-BILL [100,120]
```

Will given an 'expression out of range' error if it evaluates to below 100, or above 120. The ranges could even be expressions themselves, they don't have be numeric. This does not affect the expression in any way and is ignored by the assembler, unless the expression is out of range. Spaces and/or tabs can be used after the expression and before the range check, and will be ignored, eg:

```
LDA FRED+23 *2    [58,70]
```

One common mistake is to use expressions brackets in the wrong place and fool the assembler, for example :

```
LDA  (FRED+2)*3
```

Would give an extra characters on line error, as the expression in brackets would be treated as a indirect memory reference with Y, the *3 would be extra characters. If you wish to use this sort of expression then you would have to enter it as below :

```
LDA  3*(FRED+2)
```

or

```
LDA  0+(FRED+2)*3
```

Which would load A with the value at the absolute address of the expression.

3.1 The assembler pseudo ops.

Most pseudo ops have a number of different syntax, this is to allow the programmer to use source code directly from other assemblers without having to make too many changes. Any or all of them can be used in your programs, all of which will perform exactly the same task.

There are two pseudo opcode lists, the first covers the usual code related pseudo ops, while the second list is specifically for assembly listings related pseudo ops.

ASK - Will get a yes/no response from the user

ASK must be used on a line with a label. This pseudo opcode will stop assembly on pass 1, print the message following the ask pseudo op, adding this prompt, ' (Y/N)?', automatically. The user has to now press either [Y] or [N], in response to the prompt. If [N] is pressed, then the label is given a value of zero while depressing [Y] will assign a value of one to the label. Note: due to the way the assembler works, any text following the command will be converted into upper case. To use lower case as well put a single quote at the start of your message. For example:

```
CHEAT       ASK "Cheat mode
```

Will stop assembly and display the following message in the command window:

```
Cheat mode (Y/N)?
```

Then wait for a Y or N response.

Similar pseudo opcode is QUERY.

BANK - Downloads code into different memory banks

Syntax: BANK <*bank number 0-255*> [,*start address*]

This is a very complex command in PDS, but it allows the user to program computers which either have banked RAM, or a RAM capacity greater than 64K. If PDS finds a BANK command in your program, the indicated *bank number* is sent to the target computer. This will then select the correct bank configuration, download all the assembled code before the BANK pseudo opcode. Only the code assembled from the last occurrence of a BANK pseudo opcode, or the start of the program, will be downloaded. If a *start address* is specified, then the downloader will jump to the address, and continue the assembly.

The bank numbers mean nothing to PDS, these are just passed directly to the target computer download software, the programmer must add their own routines to the download software to interpret them. On a Commodore 64, the user could simply store the number at address 1, to select different memory maps. See the chapter on the download software for more information.

By allowing an optional *start address* the user can make the target computer jump to the routine just downloaded, while PDS continues assembling. Here is one possible use for this powerful function.

If the download software is located at $8000 and the programmer wants to write code to this address, to save having to make a new copy of the download software, simply download it somewhere else in memory. The program would then look something like this:

```
            ORG $7000

** DOWNLOAD SOFTWARE CODE (C64.DL1 for example) **

            BANK 0,$7000
            ORG $8000

** PROGRAM CODE **

            END
```

What this will do is assemble a new copy of the download software at $7000, jump to it, then download the main program at $8000, where the old download software used to be. This is assuming that the download software would take BANK 0 as being main memory. This is only one example of bank's use. It can also be used to download code overlays (routines that run at a common address) individually, then save them to disk, compress them, or move them about in memory.

Note: After a bank command, the assembler is 'reset', apart from the symbol table, which means the user will have to insert a new ORG before code following the BANK pseudo op. The SKIP function will also be disabled.

If required to write code for a computer with a number of banks the program would look something like:

```
            ORG START
**  PROGRAM FOR BANK NUMBER 1  **
            BANK 1
**  PROGRAM FOR BANK NUMBER 2  **
            BANK 2
**  PROGRAM FOR BANK NUMBER 3  **
            BANK 3
            END
```

Note that PDS will automatically download the code on pass 2, without prompting 'Download (Y/N)?', otherwise downloading could become very tedious, particularly when using a number of banks. If an error occurs in the second pass, while PDS is assembling the code for bank 2, you should note the code for bank 1 will already have been assembled and downloaded. The target computer will also still be setup for BANK 1.

To change bank, without downloading code, then simply insert the bank command before the ORG pseudo opcode. This

may have to be done in the last example, if START was actually in bank 1. Under these conditions the program will n run correctly, because the last block of code was downloaded to bank 3. So when the execution address is passed, it w execute code in bank 3. To get around this situation, insert the setup of the correct bank for the execution label, in t case a 'BANK 1' command directly before the 'END' in the program.

After assembling the complete program, the prompt 'Download (Y/N)' appears as normal.

Rather than have bank numbers such as 0,1,2.... most software will have odd numbers such as $27 for main memory, $ for bank 1 etc. The code will be more readable if you define equates such as:

```
MAIN        EQU $27
ONE         EQU $25
```

Then enter BANK ONE or BANK MAIN.

Note that when using the BANK command, skip is disabled, and you will see 'no skip' displayed in the command l window after assembly has been completed.

CBM - Commodore define message

This has the exactly the same syntax as DB, but any text in quotes will be given in the Commodore 64 screen codes rather than ASCII. For example:

```
CBM "CAT"
```

produces : 3,1,$14

This command does not allow text to be used in expressions though eg: "C" + 128 would not be allowed.

Similar pseudo opcodes, see DB.

DB
DFB
DEFB
BYTE
EQUB
.BYTE
.ASCII
.TEXT
TEXT
ASC
STR
DEFM
DM
DFM

- Defines bytes and messages.

Following the command there can be any number of parameters, separated by commas, each of which must be an expression within the range -128 to 255. Strings of text and text mixed with numeric expressions can also be used. For example:

```
DB 23,45+52-FRED,"This is a test message",13,10,"!"+128
```

To include the quotes character inside strings, put two quotes one after the other eg: "Say ""HELLO""" would produce : SAY "HELLO".

.BYTE is slightly different in that if there are no parameters it will default to .BYTE 0.

For similar commands see CBM, DC and STRING.
For a faster way of storing data see HEX.

DC - Define message, ending in bit 7 set.

This pseudo opcode is very similar to DB, but any text strings will have bit 7 set on their last character. Many programmers use this format for text, to indicate that the end of a string has been reached, for example:

```
DC 2,"cat",13,"!"
```

would produce:

2,$63,$61,$F4,13,$A1

For similar command see DB, CBM and STRING.

DH
DHIGH - Define high bytes only

Is similar to DW, but adds a '<' (gets high byte of expression only) in front of all the expressions. It is mainly used when you have tables of high and low bytes in programs. This is a common occurrence in 6502 code. For example:

```
DH 65500,23,500
```

will produce: $FF,0,1

For similar commands see DL.

DL
DLOW - Define low bytes only

This functions in a similar way to DW, but a '>' (symbol to get low byte of expression) is added in front of all expressions. This is mainly used when you have tables of high and low bytes in programs. This is a common occurrence in 6502 code. For example:

```
DH 65500,23,500
```

will produce: $DC,23,$F4

For similar commands see DH.

DO - Used in DO...LOOP or DO...UNTIL expressions

There are two different syntaxes for the 'DO' command, both uses produce blocks of similar code.

DO *expression*

***** program code *****

LOOP

This will repeat the block of code between the DO and LOOP pseudo opcodes, *expression* number of times. For example:

```
        DO 8

         LDA (FRED),Y
         STA BILL,X
         INX
         INY

        LOOP
```

Note that the expression after the DO is only evaluated once, when the DO is first assembled. When LOOP is executed the local labels are cleared, allowing local labels to be used in DO structures, ie:

```
        DO 8

         ADC #20
         BCC !1
         INX
!1       INY

        LOOP
```

The second syntax for the DO command, is as follows:

DO

***** program code *****

UNTIL *expression*

This is similar to 'DO...LOOP', but the *expression* is evaluated each time and will only exit when it returns a non zero value. This could be used in the following way:

```
DO

 LDA (FRED),Y
 STA BILL,X
 INX
 INY

UNTIL *>1000
```

fills memory with the code until the PC goes above address 1000. Note that * = 1000 was not used, just in case the code stepped 'over' that address, in which instance it would keep looping until it tried to wrap round at 65535.

DS
DFS
DEFS
.BLOCK

- Define a space of memory

Syntax : DS *length of block* [*filler byte*]

Is used for either leaving or filling areas of memory. By entering :

```
DS 20
```

the PC will be moved on 20 memory locations. Note when downloading the code the 20 locations will be filled with random data, you cannot assume anything. If you wish to fill the 20 bytes with zeros then you should use

```
DS 20,0
```

DW
DFW
DEFW
.WORD
- Define words

This is similar function to DB, but defines word rather than byte values. There may be any number of parameters following the command, all separated by commas, but will only accept strings one or two characters long. The expressions should range from -32768 to 65535. As with .BYTE, .WORD is different in that if you don't have a parameter following the command, it will default to the equivalent of DW 0.

Example:

```
DW 0,23,23*5,FRED+45,"ab"
```

For a faster way of storing data see HEX.

END
.END
- End of your program

Normally the PDS assembler will stop at the end of file even while assembling a program. By inserting an END staten in the code, the assembler will stop at that point. Specify the execution address of your program by placing it after end, for example:

```
END START
```

or

```
END $8000
```

Once the program is assembled an execution address is given. If it is downloaded, the target computer will automati jump to this address. If an execution address isn't included and you download code after assembly, the target comp will continue executing the download software. You may also specify the execution address with other commands EXEC), you may only specify the start address once though.

ERROR - Stop the assembly with a user error.

Syntax: ERROR *message*

When the assembler encounters this line, it will terminate assembly and print the specified *message* in the command line window. Once a key is pressed, you are returned to the editor with the cursor on the ERROR line. Normally this is used in an 'IF...ELSE...ENDIF' structure when something has gone wrong, or a condition has not been met. As with ASK and QUERY, the assembler will convert the message into uppercase unless a single quote is placed at the start of the message. A terminating quote is not required, and if used, will be printed as part of the *message*.

Examples:

```
ERROR "Too many aliens!
ERROR "Coordinates were off the screen
```

EQU
EQUATE
.EQU
DEFL
=

- Equate a value to a label

The line with the EQU must have a label at the beginning. The label will be given the value of the expression foll the EQU. The expression must return the same value on pass 1 and pass 2. No undefined labels can be used expression. Therefore, a forward reference to a label would not be allowed in the expression.

DEFL and = are slightly different in that because they allow the label on that line to be used again, they are re-definable. You can redefine these labels as often as you wish, for example:

```
FRED        = 2                 ;FRED is set to 2
FRED        = FRED+1            ;FRED is now equals 3
```

This could be used in a loop. To fill 20 bytes with the numbers 0 to 20 you could do the following:

```
TEMP        = 0
            DO 20
            DB FRED
FRED        = FRED+1
            LOOP
```

Normal equates cannot be re-defined, a 'multiply defined label' error will occur if you try. Equates help to make pr more readable by giving names to arbitrary figures.

EXEC START - Define your programs start address

Syntax: EXEC *expression*

This will define the execution address of the program. By using an execution address, after code is download, the target computer will jump to it. If an execution address is not specified then the code will still be downloaded but the target computer will continue running the download software. You may also define the start address by having it as an *expression* after the END command. If you wish to trace programs, or use the monitor, don't put a start address for your programs.

FREE - Adds up free space in your programs

Syntax: FREE *expression*

At the beginning of assembly, the assembler clears the free memory variable, and every time a FREE command is issu the *expression* following the command is added to the free variable. At the end of assembly the free memory is displa in the command line window. This need not be used for free memory, it could be used to tell you the value of a syml or one particular address. To use free to add up all free memory, insert a FREE pseudo opcode before each ORG, i

```
FREE $8000-*
ORG $8000
```

If the PC was on $7FF2 before the ORG then FREE will tell you there are 14 bytes of free RAM. Normally a FR pseudo opcode is inserted before all ORGs, and a free at the end of the program with RAMTOP-* as the expression. will then know the exact number of bytes available as free memory. If you don't want to use FREE to add up free mem then the function has other uses, ie:

i) Finding the length of routines by having a FREE END-START in your code. you change the routine you don't need to have list on to find out the routi exact length.

ii) Displaying the value of labels, whose values can then be used in the monito having FREE DATAAREA. The value of DATAAREA is printed a assembly.

HEX - Define hex bytes

Syntax: HEX xx.....

PDS often uses HEX statements, because they are very fast, compact and easy to read. HEX is followed by 2 digit HEX bytes, which may be separated with spaces or tabs. You can fit up to 57 HEX bytes on one 120 character line. We recommended that you put all your data in this format, simply because it's easy to use. If you have many DEFB's, assemble and download them, then upload them into HEX bytes using the monitor.

IF - Conditional assembly

Also covered in this section

ENDIF
ELSE
IFS
IFF

When assembling something depending on a condition, use the 'IF' command. If the expression after the IF returns a zero value, then the code after the IF will be assembled. If the expression is equal to zero, the code will be ignored an ENDIF is found. You may nest IF statements to a depth of 10. For example:

```
        IF FRED=23
This code will only be assembled if FRED=23
        ENDIF
```

You may also use ELSE to allow the assembler to assemble an alternate piece of code, if the expression is equal to eg:

```
        IF FRED
This code will be assembled if FRED<>0
        ELSE
This code will be assembled if FRED=0
        ENDIF
```

IFF is similar to IF, but will only assemble the following code if the expression after the IFF contains a forward refer This is necessary because normal IF statements cannot have forward references in their expression as the assembler to know the route of action on both pass 1 and 2. An error will be given if it cannot evaluate the expression compl For an example of IFF see the example macros chapter.

IFS is similar to IF, but is used for string comparisons. The code following will only be assembled if the two strings cont in square brackets are equal. For example:

```
        IFS [FRED] [BILL]
this will not be assembled
        ENDIF

        IFS [FRED] [FRED]
this will be assembled
        ENDIF
```

IFS is mainly used in macros, with parameters such as:

```
        IFS [@1] [SCREEN]
```

You may use it to see if a macro parameter exists eg:

```
        IFS [@1] []
```

You may compare the first string against a number of different strings. If any one of them is equal, the following co be assembled. For example:

```
        IFS [FRED] [BILL] [JOHN] [TONY] [FRED]
This will be assembled
        ENDIF
```

Note: Due to the way the assembler works, all the parameters will be converted into upper case.

INCLUDE .INCLUDE FILE - Reads source code from the disk. file

Syntax: INCLUDE *filename*

When PDS finds an INCLUDE pseudo opcode in the source, it will try and open the specified file and start reading source code from that file. When the end of the disk file is reached PDS will go back to the line directly after the INCLUDE in memory. The *filename* may contain drive specifiers and path names, for example:

```
INCLUDE A:\PDS\GAMES\EMPIRE\GRAPHICS
INCLUDE FRED
INCLUDE BILL.TXT
```

You may not nest include files, but you can have as many as you like in your main program. If there are large amounts of data in the program, preferably with no forward references (graphics data is a perfect example), store them in include files. As data in include files is only read in on pass 1 (with skip working), on a hard disk INCLUDE's are very fast, so assembly is hardly slowed down. This means the largest program that PDS can assemble is determined by the size of the disk drives attached to the host system and the size of source in RAM.

MACRO
.MACRO
- Define a new command in the assembler

Also covered in this section :

ENDM
.ENDM
EXITM

This pseudo op is used to generate new instructions for the assembler. To define a new instruction called FRED, will insert four NOPs into the object code, the following code would be used:

```
FRED        MACRO
            NOP
            NOP
            NOP
            NOP
            ENDM
```

This should be placed near the start of the program, before the macro is invoked. Whenever assembling FRI normal instruction, four nops will be inserted into the code instead. You may have up to 250 different macros. Y not have macros with the same name as pseudo ops or instructions already in the assembler.

Parameters may also be passed macros. Parameters should be separated by commas and are referred to in the definition by @1 to @9. So if you invoked the above macro by using:

```
            FRED bill,23,"hello"
```

Then @1 is set to 'bill', @2 is set to '23' and @3 is set to '"hello"'. Note that macro parameters need not be expr They are text strings that are inserted into the code in the macro in place of the equivalent @ number (@1 to @' example:

```
SUM         MACRO
            LDA @1
            @2 #@3
            STA @1
            ENDM

            SUM FRED,ADC,3
            SUM FRED,EOR,23
```

You may also define labels eg:

```
VAR         MACRO
@1
VARS        = VARS+@2
            ENDM

VARS        = $8000
            VAR FRED,2
            VAR BILL,1
            VAR JOHN,3
            VAR JANE,1
```

The above macro is used to define variables in a data area, when each variable has a different length. This is variables need not be tied to absolute memory addresses. It is possible to insert or delete variables as they w overlap. Also you can define labels within your program, when you need them - you do not need to define them

at the start. In the above example the labels would have been defined in the following way:

```
FRED        : $8000
BILL        : $8002
JOHN        : $8003
JANE        : $8006
```

A macro expression can be evaluated before it is inserted into the macro text, by proceeding the parameter number with '@@' as apposed to the normal '@'. This is particularly useful if you wish to define a number of new labels within a macro., ie:

```
FRED        MACRO
VAR@@1      EQU *
            ENDM

            FRED 3*2
```

This will create a varaible called 'VAR6' within the macro FRED, while a normal macro parameter would have tried to create a label called 'VAR2*3'.

If macro's parameters are all expressions, then you can refer to them as labels within the macro:

```
FRED        MACRO JOHN,BILL,JANE
            LDA JOHN
            ADC BILL
            EOR JANE
            ENDM
```

This has been included for compatibility. Assembling the macro with list on and list macro produces code for FRED 2,3,4 as follows:

```
JOHN        = 2
BILL        = 3
JANE        = 4
            LDA JOHN
            ADC BILL
            EOR JANE
```

Macros can be nested to a depth of four, but may not be recursive. Labels can be used in macros, although it would be best not to define any, as using the macro more than once will give a multiple definition error. To define labels in macros, use local labels. This will not affect other local labels of the same name in the source code, or in other nested macros. See the following example:

```
BILL        MACRO
!1          NOP
            JMP !1
            ENDM

FRED        MACRO
!1          BILL
            JMP !1
            ENDM

!1          FRED
            JMP !1
```

The code produced is quite useless, but you can see that the local labels really are local. Note that because local labels are local to macros, you cannot reference a local label outside the macro definition. For example, using a macro parameter of !1 means that the assembler would only search the macro definition for the !1. It would not use the !1 from the source code that called the macro.

Macro parameters are separated by commas, leading and trailing spaces are not passed to the macro. If you wish to
the comma character then enclose the parameter in square brackets, for example:

```
                FRED 23,[45,56,67],98

@1              = 23
@2              = 45,56,67
@3              = 98
```

EXITM is used with IF, IFF or IFS statements inside a macro. When EXITM is executed, the macro definition halt
returns to the main program - basically it is the same as an ENDM, except that uncompleted conditional structur
cleared. For example:

```
FRED            MACRO
                IFS [@1] []
                EXITM
                ENDIF
                ADC @1
                ENDM
```

This example will produce no code if no parameter is given, else it will add the parameter to A. See the example n
chapter for more useful examples.

ORG
.ORG - Setup the current PC.

Syntax: ORG *new PC* [,*true PC*]

The ORG pseudo opcode defines a new assembly address, and updates the program counter accordingly. You may have up to 100 orgs in your program. The *new PC* must not be a forward reference, it shows where the code will be assembled to in memory. If you have a *true PC* parameter, then all assembled code would use this value of the PC when the code references the PC, but will actually insert the code at the memory pointed to by the *new PC*. When downloading the code to the target computer, PDS will download the code in blocks, from the start of an org, to the end of a block of code, then from the next org, and so on. If the program comprises two small sections of code, one in low memory, the other in high memory, then PDS's download software would only download the two sections of code, not all the memory in between.

For example:

```
ORG $8000,$100
```

Would place the code in memory at address $8000, but the code would only run properly when moved to address $100. This is useful for producing disk overlays, or routine modules that move in memory. To remove the 'PC offset' just use another org, without a *true pc* parameter. If you wanted the code to continue after the relocatable routine at $8000 you would have to use something like:

```
ORG *-$100+$8000
```

Now the rest of the code would follow on in memory, and could be run where it was.

If employing a number of ORGs in programs, there is always the risk that at some point you will have code that overlaps other pieces of code already written. PDS will not trap this as some people need to use memory dynamically. The only way you can tell if you have overlapping code is when you have skip on. Instead of giving you a skip value, the display will read 'no skip'. If this ever happens, then go through all the ORGs in the code and find the one that overlaps other code.

You may ORG anywhere in memory, from 0 to $FFFF. ORGs do not have to be sequential, they can be above or below one another.

QUERY - Will get an expression from the user

QUERY must be used on a line with a label. It will temporarily stop assembly on pass 1, display any message after QUERY pseudo opcode, followed by a flashing cursor. The user may then enter an expression or number, press e and assembly will continue. The label on the QUERY line will then be assigned the value of the expression. Note due to the way the assembler works any text following the command will be converted into upper case. To use lower c put a single quote at the start of your message, for example:

```
LIVES       QUERY "How many lives do you want :
```

Labels may be entered as part of the expression, so you can enter string responses to your questions by defining a equates as below:

```
COMMODORE   EQU 1
BBC         EQU 0
FRED        QUERY "Which machine is it for :
```

If the user enters COMMODORE, FRED will equal 1, or BBC and FRED will equal 0.

A similar command is ASK.

RADIX
.RADIX - Setup the default radix for the expression evaluator.

Syntax: RADIX *expression*

The radix pseudo opcode defines the default base for the expression evaluator. Any value, from 2 to 16, can be used with the RADIX statement. The default base is 10. If using a large amount of binary data, rather than putting % infront of all the data, you could insert a RADIX 2 command before the data, and a RADIX 10 command after it.

For example:

```
RADIX 2
DB 10101101,10101010
DB 00101010,10101010
RADIX 10
```

Note: The expression following the RADIX pseudo op is ALWAYS in base 10 (the default).

REPEAT - Repeats the following line a number of times.

Syntax: REPEAT *expression*

Repeat is used for repeating one instruction or any line a number of times. For example:

```
REPEAT 4
LSR A
```

This is equivalent of inserting the following in your code:

```
LSR A
LSR A
LSR A
LSR A
```

SEND
DLOAD
DOWNLOAD

- Tells PDS where to send the program once assembled.

Syntax: SEND *destination*

The SEND pseudo opcode is used to determine the destination of the object code, once a program has been assembled. The *destination* can be one of the following:

SERIAL	- Via the serial port at 9600 baud
SERIAL1	- Serial, 19200 Baud
SERIAL2	- Serial, 1200 Baud
SERIAL3	- Serial, 300 Baud
COMPUTER0	- Computer 1 port, 'nibble' protocol
COMPUTER1	- Computer 1 port
COMPUTER2	- Computer 2 port

If employing this command in your program, when assembly is completed you will be prompted 'Download (Y/N)?'. If you hit any key other than [N] then your code will be downloaded to the specified destination. Once the code has been downloaded, the prompt remains on the screen. You may resume editing by hitting any key other than [Y]. Pressing [Y] will download your code once again. While code is downloading the cursor will flash. If it seems to be taking too long, then there is probably a fault in either the download software on the target or the interface connections. You may press [FINISH] at any time, to break out of downloading. In all other areas of PDS there is an automatic time-out while downloading, but the assembler does not do this to allow you to get the target computer ready to receive the code.

The interface card inside your machine has two ports on it. You can download to different computers attached to these by specifying COMPUTER1 or COMPUTER2. Some computers such as the BBC Micro need a special form of download software, which sends and receives 4 bits at a time rather than whole bytes. To send in this format, use 'SEND COMPUTER0', with the target computer plugged into port 1. The send using the serial ports only needs to be used with computers that do not have PDS interface available for them, but the protocols remain identical. The data is sent with 8 bits, 1 stop bit and no parity. You should go at the fastest reliable speed. Some machines are capable of 9600 baud, most are better at 1200. Not many can keep up with 19,200 baud. If you cannot get any results, even at 300 baud, then you must either have a faulty RS232 cable, or you have not setup the target computer properly.

SKIP - Used to speed up assembly time

Syntax : SKIP *expression*

This command is specific to PDS, and enables your programs to assemble at nearly twice the speed with no drawb
The command makes a note on pass one, of any large sections of program that doesn't need to be reassembled aga
pass 2, these are then SKIPped over on pass 2. As a large amount of information has to be stored for every skip, th
a limit to how many 'skipping' points PDS can remember. Therefore, after assembly, the percentage of skip table u
displayed in the command line window. If the skip percentage returned is 100%, then the skip table has been fill
early on in the program and has probably missed some blocks towards the end of the program. The best skip table
is about 80% to 95%. There is an *expression* after the skip command, for average length programs this will range
about 15 to 30. This *expression* tells the assembler the minimum size of a block which contains no forward reference
can inserted into the skip table. So the lower the number, the larger the number of blocks that are added to table.
number is too high, very few or maybe no blocks will be found. If you have to use include files, try to have just d
them (no forward references), that way they will only be read in once on pass 1, on pass 2 they will be skipped. I
ever displays 'no skip' after assembly, then there is an ORG overlapping with a block of code previously coded, or y
using the BANK command which disables skip. Skip must be placed at the start of programs, before any ORGs.

STRING - For defining strings ending in zero.

This has the same syntax as DB, but any text strings will have a zero byte added after the last character. This is similar to the DC pseudo op. Many programmers use a zero to define the end of a message stored in memory.

Examples:

```
STRING 13,"Cat","dog"
```

produces :

```
13,$43,$61,$74,0,$64,$6F,$67,0
```

2 Assembly listing related pseudo opcodes

ere are a number of pseudo opcodes specifically designed for listing assembled text. These have been separated from main pseudo opcode list, as we felt they needed to be explained together.

IST
IST
ST

l the above commands have one of three parameters, ON, OFF or MACRO. LIST ON and LIST OFF control the output to the screen. When 'LIST ON' is selected, you will see the source code displayed in the current window, ... it is assembled on the second pass. The listing produced is similar to the sample listing below:

```
0000:                   ORG 10
000A:  00               NOP
000B:  00               NOP
000C:                   END
```

ıe first HEX number at the start of each line shows the current address of the program counter, and any code on that ıe will be assembled to this memory address. The character directly following the HEX number shows the action taken the assembler for that particular line. A colon (:) signifies that the line has been assembled. A '+' means that the line ıs been ignored, this would occur if an 'IF' structure had failed at that point. The only other alternative is a '>' character, is means that a macro definition was decoded at that particular line. The next list of HEX bytes represent the value of e bytes that were assembled for that line, a maximum of 5 bytes can be displayed at that position. If an 'IF' or 'EQU' atement was assembled on the line, then the result of expression is displayed instead, this allows you to see what the xpression was evaluated to in the listing. An example of this is shown below :

```
000E:  0017=            FRED        EQU 23
```

this case fred was assigned the value 23, so at the start of the line 0017= was printed.

Vhen listing to the screen any character going off the edge of the screen will be lost.

ormally when you list a macro call, the assembler will print something like this:

```
000F> 00                CAT
```

his is a macro called cat, which produces a single zero byte. If you wanted to see exactly what the assembler was assembling a the macro, you could turn LIST MACRO on. The default is off, by using the instruction LIST MACRO, you toggle the ption on and off. With LIST MACRO on, the above example could look like this:

```
000F>                   CAT
000F> 00                NOP
0010>                   ENDM
```

his macro is very simple, in fact all it does is insert a NOP, but you can now see the assembler's action.

istings to the screen do not use most of the list options (see LSTOPT), but are paged, so that once a page is printed, a ey has to be depressed before the next page is displayed. For a very fast continuous listing, just hold down a key.

LLIST
LLST

Are the same as LIST, except they output the listing to the printer. You may print to either screen or printer, or both if you wish. Note that LIST MACRO applies to both printer and screen, for example, LLIST MACRO will not be accepted. There are many commands for configuring the printer, while printing listings. These are:

PRINTER

This works in the same way as define byte (DB), but all the parameters are sent to the printer. This allows you to setup the printer with specialized control sequences, for example:

```
PRINTER 27,"@",12
```

Would initialize and do a form feed on Epson compatible printers.

SKP

This pseudo opcode needs just one parameter, and sends the specified number of line feeds to the printer, ie:

```
SKP 5
```

This would send five line feeds to the printer, and would leave five blank lines in your listing.

PAGE

This command has a number of different syntaxes, these are:

PAGE **(no parameters)**

Will issue a form feed to the printer, moving the paper to the next start of form.

PAGE *expression* **(parameter 15 to 255)**

Will define form length for the printer, ie the number of lines per page.

PAGE *expression , expression*

The first *expression* defines the maximum line width, and the second *expression* defines the form length.

PAGE "nnnnn

This will issue a form feed to the printer, then define the current title for the code listing. This is equivalent to an EJECT and TITLE used together. Note that you need an initial quote in front of the text expression, otherwise the rest of the text will be converted into upper case.

EJECT

Will issue a form feed to the printer, ie move the paper to the top of form. This performs exactly the same task as a PAGE instruction by itself.

TITLE "nnnn
TTL

This defines the current title, which will be printed at the top of each page (centered) from this point on. This can also be achieved by using a PAGE "nnnn instruction. Note that the title will be converted into upper case unless a single leading quote is included in the string expression.

SUBTITLE "nnnn
STITLE
SUBTTL

This will define the current sub-title that will be printed at the top left of each page. Note that the title will be converted into upper case unless a single leading quote is included in the string expression.

WIDTH expression

This will define the maximum line length. Any line longer than this value will either be split or truncated, depending on the current list options (see LISTOPT).

LISTOPT
LSTOPT

Syntax: LISTOPT *expression*

This will set and reset the list option flags. The specified *expression* will be XORed with the current option flags. So LSTOPT 5 followed by LSTOPT 1 is exactly the same as using LSTOPT 4. There are 16 different list options, each bit in a 16 bit number performs a single task. Therefore, most people define list options in the following way:

```
LSTOPT 1024+512+4+2+1
```

This is easier to understand, and use, than working out the exact 16 bit number manually, as each used can be easily distinguished. The full range of list options follow:

BIT	FUNCTION
1	Line numbers printed at the start of the line
2	Add time and date to the top of pages
4	Add version number to top of pages
8	Add a page number in the top right
16	Add a page number in the bottom left
32	Print file headers from the RAM source
64	When include files are started print a small header
128	Print printer pseudo ops (eg: PAGE, TITLE...)
256	Print symbol table in PC order (else alphabetically)
512	Remove $ or & from the end of the PC value
1024	Don't do pages, just continuous listing
2048	Truncate text at the end of line
4096	Same as LIST MACRO when enabled
8192	Add a file number in the top right corner
16384	When text wraps, print 3 tabs, (else just a ;)
32768	Global ignore all list pseudo ops

When a particular bit is set, then that function is selected. LISTOPT defaults to a value of 0.

Most functions are quite obvious and change the format of the pages being printed. The following require a little explanation:

Option 1 — These are line numbers that can be referenced using editor functions. This also means that line numbers re-start from 1, at the beginning of any file, either a memory file or a disk include file.

Option 32 — Will issue a form feed then print the PDS start of file header when the start of a file is printed. Otherwise there will be no break in the source code as the assembler moves from file to file.

Option 64 — Allows you to see which file is being included.

Option 128 — Will make PDS print all the printer pseudo ops to the printer - these are normally not sent to the printer, as list pseudo ops only change printer formats and do not generate any code.

Option 32768 — When bit 15, or 32768 is set, all following printer commands (including LSTOPTs) will be ignored, until another LSTOPT is executed that flips bit 15 (32768). So if you globally wanted to ignore all printer commands you could do:

```
LIST OFF
LSTOPT 32768
```

Now, whatever you try, even LIST ON, will be ignored. When requiring to ignore all the printer commands and just get a listing use:

```
LIST ON
LSTOPT 32768
```

4.1 Example programs

PDS is very flexible in allowing many different syntaxes for most pseudo opcodes. Although there are one or two commands that you will find in every program, below is an example 'shell' of a PDS program :

```
            SKIP 20                 ;Turn skip on for speed
            ORG $8000               ;Must be after skip
```

** program source code **

```
            SEND COMPUTER1          ;Download after assembly
            END start               ;End source and give program execution a
```

The minimum source you need in a your source code is an ORG, otherwise PDS will not know where to assemble the source code. Without the SKIP function, assembly time will increase, without the SEND pseudo opcode no download prompt will be given after assembly, without the END pseudo opcode, assembly will only terminate at the end of file 7.

Below is an example program for flashing the screen on the Commodore or BBC.

```
;
;
;
            ORG $7000
            SEND COMPUTER1
;
COMMODORE   EQU 1
BBC         EQU 0
;
COMPUTER    QUERY "Enter computer for (BBC, COMMODORE) :
;
START       LDY #0

            IF COMPUTER
SCREEN      EQU $400

            ELSE                    ;Work out screen start

SCREEN      EQU $7C00
            ENDIF
;
```

```
BEGIN       LDA #32              ;Fill screen with spaces
            JSR FILL
            LDA #127             ;Fill screen with blocks
            JSR FILL
            DEY
            BNE BEGIN            ;Repeat 256 times
            RTS                  ;Return to downloader

FILL        LDX #0
!1          STA SCREEN,X
            STA SCREEN+$100,X    ;Fill 1024 bytes
            STA SCREEN+$200,X
            STA SCREEN+$300,X
            DEX
            BNE !1
            RTS

            END START            ;Jump to START after downloading
;
;
```

When assembling this program, you are prompted to enter either BBC or COMMODORE. You will then, almost instantly, be asked if you wish to download, if you press any key other than 'N' then it will be downloaded and run the program on the target computer. The screen should flash for a few seconds, then return to the download software with the screen totally filled.

4.3 Example macros in the PDS 6502 assembler

Macros are very useful, they can save typing, memory, and improve source code readability. Here are a few sample macros that you can try to understand, or even use.

```
JCC             MACRO
                IF (@2-*<130) ! ($-@2<126)
                IFF @2
                ELSE
                B@1 @2
                EXITM
                ENDIF
                B@1 !1
                JMP !2
!1              JMP @2
!2
                ENDIF
                ENDM
```

This macro expands the 6502 instruction set. Instead of entering 'Bcc *addr*' (where cc could be a condition, ie NE, CC, EQ, CS etc.), and being unsure if the branch can be made, it will insert a branch if a branch is posiible, otherwise a jump is inserted instead. Note that this will always use a jump if you have a forward reference to a label, simply because it cannot know how far away the label might be. For example:

```
FRED            NOP
                JCC EQ,FRED
```

Will generate a BEQ FRED, and

```
FRED            DS 300
                JCC NE,FRED
```

Will generate :

```
                BNE !1
                JMP !2
!1              JMP FRED
!2


ZVAR            MACRO
@1              EQU ZLIM
ZLIM            = ZLIM+@2
                IF ZLIM>255
                ERROR "Out of zero page work space with variable @1
                ENDIF
                ENDM
```

This macro allows you to define zero page variables automatically. An error will be generated if you try to use work space above $FF, as this is no longer zero page, and could start to corrupt the stack. An example of its use can be seen below:

```
ZLIM        = 2
            ZVAR XVAL,2                 ;Allocate 2 bytes for XVAL
            ZVAR YVAL,2                 ;Allocate 2 bytes for YVAL
            ZVAR TEMP,1                 ;Allocate 1 byte for TEMP
            ZVAR TABLE,250              ;This will generate an error as
zero page
```

These types of macros are useful, because you never need to allocate memory addresses yourself, which means they can change automatically. This also prevents you from having overlapping variables, which is a common programming error.

If using many ORGs in programs, you find that they often overlap, ie 'no skip' is displayed at the end of assembly. You can use a macro to perform all the program ORG's, and trap any overlapping, ie:

```
CORG        MACRO
            IF $@1
            ERROR "Overlapping code at address @1
            ENDIF
            ORG @1
            ENDM
```

Now instead of ORG, use CORG address and your code will be checked for overlapping sections automatically. This macro is also useful when using banked RAM, as SKIP is then disabled and will not indicate overlapping code.

5.1 The PDS Download Software

The download software has been written over a period of time, and has been heavily optimised for speed and size. All the different types of download software are supplied on the master disk, with all the PDS software. There are three main types of download software:

Short 'dumb' download software (*.DL0)

This is the shortest possible software that allows you to download and execute code. This should be used when free memory on the target machine is restricted. It will not operate correctly with most monitor commands, such as modify, although commands such as fill, that only download data, will still operate. This version is also useful when writing new download software, or the software has to be typed, in an emergency.

Long 'clever' download software (*.DL1)

When you purchase a target computer interface, this version of software is supplied on tape or disk. It will allow you to use all the monitor's features (except analyze which requires the DL2 version download software), while still allowing you download and execute a program from the assembler.

Interrupt driven software (*.DL2)

This is the download software that runs under the interrupts and allows you to use most monitor commands while your program is running on the target computer. This allows you to watch areas of memory change, stacks building down, or modify variables while a program is running. You may not alter registers or trace from the monitor, as this would cause data areas to clash and not be of much benefit. It is useful to insert the source code of the downloader within your own program while it is being developed - this make debugging faster and easier.

All the download software can be positioned anywhere in memory, and run completely independently, with all data areas self contained. You are free to rewrite them if you wish, and mould them to suit your own requirements. They can be made substantially faster by removing the CALLS to the GETBYTE and SENDBYTE routines in the upload and download loops, and replacing them with an exact copy of the actual routines.

DL0 and DL1 both have the interrupts disabled, this allows you to download programs over operating system workspace, or over interrupt vectors, without crashing the target machine.

When downloading programs, ensure that you do not download over the downloader itself. This applies even when you are downloading the download software itself, as self modifying code is used in all versions of the software. Although this sounds obvious, you will be surprised the number of times this will happen to you, and could leave you with mystifying crashes.

Most users load their download software of tape or disk, into a completely free area of memory. This will then allow them to return to the downloader at any point, to download new code, so speeding up development greatly, and saves having to continually reload the download software. On the Commodore 64, $C000 is normally a good location for the download software, while $1900 can be used on the BBC. Some 6502 programmers position the download software at the bottom of the stack, this will save memory and keep the download software out of the way.

When you first setup the PDS, it is normally best to make up your own version of the download software, and save it to tape or disk. To do this load the correct download software into PDS (either C64 .DL0 or DL1), change the ORG at the top of the source, to the required address (remember, avoid overwriting the present download software). Load the supplied download software into the target computer and assemble and download the new download software. You should now save it to disk or tape.

5.2 The PDS download software protocols.

There now follows a complete description of the 6502 communication protocols used by PDS software. The software is designed to be machine independent, so the same protocols can be used, regardless of the target machines processor or make. All the communication is basically the same, there are four major routines, SEND BYTE, GET BYTE and two direction swapping routines. These have been optimised as much as possible and all the data formats are as compact as possible. For example, if you had a small piece of code in low memory and a small piece of code in high memory, then PDS would automatically download two small blocks, instead of downloading the memory in between as well. There now follows a description of the main parts of the downloading system on the target computers.

The interface hardware

The target machine interfaces are quite straight-forward. They have eight bi-directional data lines, which are buffered, one input flag and one output flag, which are also buffered.

Initialization of the download software

The first thing the software must do is to setup the direction of the eight data lines, these should start inputting from the main computer. The output flag line must be set to 1, when the machine is first turned on, this line should not be toggled on initialization.

The main loop of the download software

At the start of the main loop, you must continually read in bytes from the interface, until a command byte is received (from 180 to 187). The correct routine must then be entered, to perform the required function. PDS may sometimes send a 179 padding byte, this should be ignored by the downloader, and is only used to make sure that all blocks of data are an even number of bytes long. This ensures that the handshaking lines are always in the same state at the start of each function. Each command byte is now explained:

Command 180 - Download code

This will download an area of memory from the main computer into the target computer. This command is present in all download software and is used by the assembler for downloading programs. Note that programs will be downloaded in blocks, from the start of one ORG, to the beginning of the next or the end of the program. Monitor commands such as fill, use this command. After the 180 command byte, the start and length addresses of the code are downloaded, in that order. All addresses are downloaded high byte first, followed by the low byte. A length of zero, is actually a length of 65536. Once the length is sent, the block itself is then downloaded into the specified memory addresses. The downloader now re-enters the main loop. For example, to download four bytes (255,65,23,45) into address $8000, the following command block would be sent to the target computer:

179	- Padding byte
180	- Command, download
$80	- Start address, high
0	- Start address, low
0	- Length, high
4	- Length, low
255	- 1st data byte
65	- 2nd data byte
23	- 3rd data byte
45	- 4th data byte

Command 181 - Jump to an address

This command will download an address from the main computer and 'JSR' to it. If an RTS is executed, then control will return to the main loop. This command is also found in all versions of download software and is used by the assembler if you specify a start or exec address for the program. The main computer will always send a padding 179 byte, as there are only three bytes in the main command. For example, to execute a routine at address $1234 the following list of bytes would be sent:

179	- Padding byte
181	- Command, Jump
$12	- Start address, high
$34	- Start address, low

Command 182 - Upload memory

The DL0 software does not support this command. It is similar to the download (180) command in format, except that after the length is sent, the ports are reversed and the requested block is sent up to the main computer. Once the data block has been sent, the ports are once more reversed, and the main loop is entered. For example, to upload three bytes from the target computer, starting at address $1234, the following would occur:

179	- Padding byte
182	- Command, upload
$12	- Start address, high
$34	- Start address, low
0	- Length, high
3	- Length, low

Reverse ports (flags act as if half a byte has been sent)

??	- 1st upload byte
??	- 2nd upload byte
??	- 3rd Upload byte

Reverse ports (Like half a byte again)

Note that the two reverses of ports act as if one byte has been sent, hence the need for the padding byte.

Command 183 - Select bank

All version of the download software have this command. The assembler uses this feature in the BANK command, the monitor uses it in the Z command. The main computer just sends the byte given in the command, unaltered and expects the target computer to act on it accordingly, in whatever way you wish the command to be used. You will have to write the code to support this command if you have a special use for it. This is a simple two byte command, 183 followed by the bank number.

183 - Command, select bank
??? - Bank number

Command 184 - Send all registers to the main computer

This is only in the DL1 version of the software. It is used in the tracing and register related functions of the monitor. A block of 26 bytes are sent to the main computer. This block contains all the 6502 registers, and is uploaded using the Upload command (182). The registers are setup for 26 bytes and the start of address of the upload is setup by the downloader itself. Bytes are sent as follows:

179 - Padding byte
184 - Send registers command

Reverse ports (same as sending half a byte)

7 data bytes are sent

Reverse ports back (same as sending half a byte)

There is no padding byte, because the length will always be an even length. The register block is stored in the following order:

P - Status register
A
X
Y
S - Stack pointer
PC - Low
PC - High

The register block is updated when the download software is first entered, when you jump out of the download software. All the registers are loaded with values from this table.

Command 185 - Get registers from the main computer

This command downloads a 7 byte register block from the main computer. The registers of the target computer will be loaded with these values when the download software is jumped out of. The register block is in the same format as the 184 code. This command is only found in DL1.

Command 186 - Trace code buffer

This is the main command used by the tracer, and is only found in DL1. Three bytes are downloaded after the command and are stored in a buffer. The registers are loaded from the register block, and the four byte buffer is executed. The registers are then saved back into the register block and the main loop entered. Note that the code executed in the buffer has to be carefully chosen - if you were to execute jumps then the download software would lose control, and function incorrectly. If the instruction you wish to execute is shorter than three bytes then the rest of the buffer must be padded with NOPS. The tracer handles this all automatically, so you need not worry, but it is important that you are aware how the system works. The tracer will emulate some instructions itself, such as jumps and just move pointers. When the tracer comes across an instruction such as 'ADC #23', then it will download 'ADC #23', 'NOP' into the instruction buffer. The buffer is then executed and the register block is uploaded, to see what was altered.

Command 187 - Return Analyze address

This command is only used in DL2 software. It will send two bytes up to the main computer, containing the address of the PC, when the interrupt occurred. The analyze command in the monitor uses this command, to work out where the program is spending most of its time.

5.3 The control lines during PDS communication

There now follows a description of the four low level routines in PDS's communication software. Note that the protocol does not set the control lines directly, but toggles the lines to communicate. All the protocols are designed so one computer can never get ahead of the other, even if there is a big speed difference. There are no error checking protocols, as the interfaces are very reliable, and error checking will slow down processes, and no real action can be taken, even if an error was detected.

SENDING A BYTE TO THE TARGET COMPUTER

i) The main computer puts the byte on the data lines, then flips its output line. The main computer now waits for the input line to change state.
ii) The target computer waits for the control input to flip, the byte on the data lines is then read in and its output control line is toggled.
iii) The main computer initiates this process first, and the target computer should finish the process first.

GETTING A BYTE FROM THE TARGET COMPUTER

i) The main computer waits for its control input to flip, then reads the byte from the data bus, and flips its control output.
ii) The target computer puts the byte on the data lines, then flips its output line. It then waits for it's input control line to flip.
iii) The target computer starts this process first, the main computer finishes first.

SWAPPING PORTS, FROM Main --> Target TO Target --> Main

i) The main computer flips its data lines direction and then its control line.
ii) The target computer waits until the control input flips and then reverses the direction of its data lines.
iii) The main computer starts this process first, and is finished first.

SWAPPING PORTS, FROM Target --> Main TO Main --> Target

i) The main computer waits for its control input to flip, then changes the direction of its data lines.
ii) The target computer changes the direction of its data lines, then flips its control output.
iii) The target computer starts this process first, and is finished first.

You many enter the monitor at any time from the main PDS editor by pressing the [MONITOR] function key. In the monitor the screen is divided into two areas. The top two lines of the screen show the 6502 registers and their values. The main area on the screen is the command area.

You may use the cursor keys to move around the command area, [HOME] will move you to the top left of the screen. [CTRL] and [HOME] will clear the screen and move you to the top left corner of the display.

If you press cursor left at the beginning of a line, the cursor will move to the end of the previous line. Commands may be entered in upper or lower case and must start in the left hand column of the screen.

You may return to the editor from the monitor by pressing the [FINISH] function key. The monitor screen will be saved, this allows you to flip between the monitor and editor without loosing any work.

To execute a command, place the cursor anywhere on the line with the command and press [return]. PDS will then execute the command, ignoring any extra characters after the command.

You do not need a target computer attached to your system to enter the monitor. If you are using the 'Clever' PDS software en all the monitor commands will function as normal. If you are using the 'Dumb' download software then only a few monitor commands will work. The others may give unusual responses or error messages.

When a monitor command fails or contains an error, PDS will put a question mark at the end of the line and beep. You may then correct the error and press enter on the line to execute it again. If there is something wrong with either the download software or the link between the computers you will get an error message such as :

```
Error, other computer not receiving
```

When you get an error like this first make sure that the target computer is running the download software. Then check the connection between the main computer and target is secure. If the error persists then you must assume that there is something wrong with the download software, or it has been overwritten.

6.2 The PDS monitor commands.

On the following pages the PDS monitor commands are listed, their syntax explained, why they may not work and examples on their use.

In all commands the expressions that you use are evaluated using the assemblers expression evaluator. This means that if you have just assembled a piece of code then you may use the labels in your expressions, for example, DSTART would disassemble from the label START. The evaluator has been expanded to allow you to access memory locations, you do this by putting curly brackets around the expression. The value within the brackets will be evaluated and used as an address. The byte at this memory location in the target computer will be returned. For example, B FRED,{FRED}+1, Would increment the byte at location FRED. You may also return a word at a memory address by using square brackets, for example, M[FRED] would modify memory pointed to by the word at location FRED. Quite complex expressions may be built up using these sets of brackets. In both cases there is no limit on the number of times the brackets may be nested.

Some monitor commands don't require any target computer attached, they are : L,O,P,Q,V,X.

Most commands need some sort of target computer attached and running the appropriate download software. There are three types of download software we supply :

DL0 - This is the shortest and can only download and run programs. It is meant to be used with the assembler when space on the target computer is at a premium. Some monitor commands will operate with this download software. They are : B (modify only), E, F, G (although it will perform a Jump), W (modify only), Z.

DL1 - This is the longest download software, about 400 bytes. It will work with all the commands, except Analyze, which requires the download software under the interrupts. This should be used if possible, else DL0.

DL2 - This is designed to be called from your interrupt routine, it will work will all the commands except R,T, as it cannot trace or alter registers. Its main use is to examine your variable area during the running of your programs, or to analyze the programs execution, or to modify data while the program is still running.

If you have the incorrect download software in the target computer, then you may get unpredictable errors, or commands may not work at all.

A - Analyze a programs execution

This can be used in three different ways:

1] A

This will repeat the last Analyze command, or A0,65535,1 if not used before.

2] A *start address* , *end address*

This will analyze between the two adresses, the whole of memory is 0,65535.

3] A *start address* , *end address* , *count value*

This is the same as 2] , but sets up the *count value* as well. The count value remains set until it is changed again. It starts being set to one.

This will give an error if:
There is a fault in the link to the target computer
The DL2 type of software is not running under the interrupts

Analyze is designed to allow you to see where your program is spending most of its time. Install DL2 download software in the target computer, running under the interrupts of your program. When your program is running type A0,65535,1 - This will display a bar graph, each bar covering about 4K of memory. PDS will print one star on a row when an interrupt occurs in that address range. To make it more accurate you may set the *count value* higher, this tells PDS how many times an address must occur before a star is printed. Once a bar reaches the other side of the display, that shows where your program is spending most of its time, now narrow the address range eg :

```
A 1000,2000
```

Keep doing this until you have an address range of about 100 bytes, this will be the most used routine in your program, optimise this one section of code and you will see significant improvements in the speed of your code. The minimum address range is 15 bytes. It will take slightly longer to produce a bar graph as you narrow the address range because it will not interrupt as often in that area of memory. The analyze command will not work properly if your program waits for VSYNC's, or other timed, as interrupts may always occur at this point in the code. Removing these waits will make this feature usable, and we are sure that you will find this command invaluable.

B - Change or display a byte at a memory location

There are two ways to use this command:

1] B *address*

This will display the byte at the address given. The value will be displayed in hex, decimal, binary and ascii.

2] B *address* , *byte value*

This will move the *byte value* into the address given.

address
This is the address of the byte to be changed

byte value
This is a byte value, from -128 to +255

This will give an error if:
The *byte value* is not in the range -128 to +255
There is a fault in the link to the other computer

1] Needs the clever PDS download software
2] Only needs the dumb download software

NOTES:

The W command has the same function, but works on words instead of bytes.
This command is mainly used for altering values in your program, to quickly see the effect of small changes.

EXAMPLES:

```
B$4000,23
```
This will load the location $4000 with the value 23

```
B FRED,{FRED}*2
```
This will double the byte value at location FRED

C - Copy block of memory

C *start address* , *end address* , *new start address*

start address
This is the first address used by the block to be moved

end address
This is the address of the last byte in the block

new start address
This is the position that the new block will start from

This will give an error if:
The *end address* is lower than the *start address*.
The new block goes above $FFFF or 65535.
The 'clever' PDS download software is not being used.
There is a fault in the link to the target computer.

NOTE:

The new block can overwrite part or all of the old block. PDS uploads the whole block into the main computer, then downloads it at the new location. This requires the download software that can send and receive blocks of memory.

EXAMPLES

```
C0,$3FF,$4000
```

This will move the first 4K of memory to memory location $4000

```
C  VARS, VARE , NEWVARS
```

This will move the memory from VARS to VARE and place it at NEWVARS onwards.

D - Disassemble memory

This command can be used in three ways:

1] D

This will disassemble a block of memory starting from where the last disassembly finished.

2] D *start address*

This will disassemble a block of memory starting from *start address*

3] D *start address* , *end address*

This will disassemble a block of memory starting from *start address* and ending with *end address*

4] D# *start address*, *end address*

The code between start address, and end address will be disassembled and inserted at the current cursor position in the editor. This is extremely useful when you wish to compare code, or examine other programmers code in more detail.

The PDS disassembler is a symbolic disassembler. This means that if a program had just been assembled, the disassembler will use them when it generates a disassembly. To use symbolic disassembly, ensure that the 'Q5' option has been correctly setup (see chapter 6.3 and the 'Q' command in the monitor for more details).

This will given an error if:
The target computer is not running 'clever' download software
The *end address* is less than the *start address*

NOTE:

The disassembler uses standard 6502 pesudo ops. When it comes across an instruction that it cannot disassemble it will print DB followed by the bytes it did not understand. For example:

```
8000: FF     DB $FF
```

If type 3] (D *start address*,*end address*) is used then PDS will disassemble very quickly, it will stop when any key is pressed, and wait until any other key is pressed before continuing, pressing [FINISH] or [ESC] will stop the disassembly.

E - Enter a program

E *start address*

start address
Is the address of the first byte of the program

This will give and error if:
There is a fault in the link to the target computer.
Only the dumb download software is needed.

NOTE:

This command is used for entering programs when you are given the program in hex or decimal. As you enter the program, PDS will disassemble it when you enter whole instructions. For example if you had the program 88,8C,03,02,60 to go at $8000 then you would enter:

```
E8000h
```

Then type 888C030260, while you are typing, PDS will print on the display:

```
8000: 88                DEY
8001: 8C 03 02                  STY $0203
8004: 60                RTS
8005: *
```

This is very useful as it allows you to check the program you are entering for mistakes.

To stop entering a program, press [FINISH]

Note that PDS only sends the bytes down to the target computer when a whole instruction has been entered.

When you are entering hex, PDS will allow you to delete characters entered wrongly. If you press delete when the cursor is next to the address then the address will be decremented by one.

When you enter programs in decimal, press enter or comma after each byte entered.

Hex or decimal program entry and the format of disassembly may be changed using the config system or 'Q' options command.

F - Fill memory with data

F *start address* , *length* , *data byte(s)*

start address
This is the address that the fill will start from

length
This is the number of bytes that will be filled

data byte(s)
This can be any number of bytes. The memory area will be filled with this pattern of bytes

This will give an error if:
The length is zero.
There is a fault in the link to the target computer.
Only the dumb download software is needed.

NOTE:

Memory can be filled with either a single value, or a pattern. For example to fill the first 4K of memory with 0 you would use:

```
F 0,4096,0
```

To fill the first 4K of memory with the repeating pattern 1,2,3,1,2,3,1,2,3..... enter:

```
F 0,4096,1,2,3
```

G - Call a routine and return

G *start address*

start address
Is the address that will be called

This will give an error if:
There is a fault in the link to the target computer.

NOTE:

This will use the download software to call the routine. This means there will be a return address on the stack. If a return is executed then it will go back into the download software. If you don't want a return address on the stack and you don't need to go back into the download routine then use the 'J' command.

This command is mainly used for testing routines. If you first use the R command to setup register values, then G*routine address* then use the R command again to see how the registers have changed. For example:

```
R X=23,Y=2
G MULTIPLY
R
```

If you only have the DL0 type download software in the target computer, then the routine will be JUMPED to, not called.

J - Jump to an address

J *start address*

start address
Is the address that will be jumped to

This will give an error if:
The 'CLEVER' PDS download software is not being used.
There is a fault in the link to the target computer.

NOTE:

This is similar to the G command, but executes a jump to the routine. This means that a return will not return you to the download software, your program will have to jump back into the download software itself. This is mainly used for jumping into the middle of a routine after setting up all the registers.

L - Loop and execute commands quickly

L

This command is used to execute a string of monitor commands one after the other very quickly. What you do is type all the commands starting in the top left of the screen, and place an L at the end of them. When you press enter on the line with the L , PDS will do a 'HOME' and keep doing returns, executing any commands it comes across. For example, clear the screen using control home and press M (return) then L (return). You will see a block of memory rapidly changing. PDS is executing the M command, then looping and executing the M command again, over and over. To stop PDS, press any key. When PDS reaches the L command it will stop looping. If any of the commands give an error, then the loop command will also be stopped.

A common use for this command is for download software running under the interrupts, while a game is playing for example. To have the command M DATAAREA looping. This will show you the data area changing as the game is being played. You could also modify the stack and see it build up and down, or the keyboard buffer.

You may build up very complex loops such as :

```
R X=23,Y={BILL}
G MULTIPLY
W [STORE],[RESULT]
W STORE,[STORE]+2
B BILL,{BILL}+1
L
```

If MULTIPLY is a routine that multiplies X by y and returns the result in a location called RESULT, then the above string of commands will build up a table of results from the address in STORE onwards. Note if you wanted to stop the above loop when a certain address in STORE was reached then you could add the following command to the loop:

```
X 1/([STORE]-ENDADDR)
```

Which would stop and give a division by zero error when the contents of STORE was the same as ENDADDR.

Loop is a command that may not be immediately useful, but becomes invaluable after a while.

K - Enter sprite grabber

K*x bytes, y pixels*

The sprite grabber is one of the most recent and powerful additions to PDS. Use the 'K" command to enter the sprite grabber, with the values *x bytes*, the width of the sprite to be grabbed in bytes, and *y pixels*, the height of the sprite in pixels. The sprite grabber is only intended for use with a bitmap screen, so before you perform any sprite grabbing, ensure that the correct target computer has been selected using the 'Q1' option, and the correct screen base address has been defined using the 'P@' option.

When the sprite grabber is first entered, the sprite cursor is positioned at the top left of the screen, at co-ordinates 0,0. The following sprite information is displayed while you are grabbing sprites:

```
Sprite size 4,16
Use cursor keys and :
Enter - Upload sprite.
Space - Invert sprite
Esc   - Return to monitor
Sprite number 00, screen address $4000, pixel position 0,0
```

The sprite cursor appears at the top left of the sprite to be grabbed, as an 8 by 1 pixel line. The cursor keys on the PC can be used to move the sprite cursor, and will update the current screen address and pixel positions accordingly. Pressing [SPACE] will invert the current sprite under the cursor, this allows you to see exactly what screen area the cursor covers. Press [SPACE] again so that the sprite reverts to its original state, before the sprite cursor is moved again. Pressing the [ENTER] key will upload the sprite into 'HEX' statements to the current text cursor position in the editor, and increment the current sprite number. The graphics editor header will also be inserted at the begining of the sprite data. The sprite is uploaded in left to right, top row to bottom row format, and can be read into the graphics editor without modification, as long as its size can be catered for by the graphics editor.

Different types of cursors can be used by the sprite grabber, these can be set using the 'Q8' option, see the 'Q' command and chapter 6.3 for more details.

M - Modify a block of memory

This command can be used in two different ways:

1] M

his will modify a block of memory starting from where the last modify command finished.

2] M *start address*

This will modify a block of memory starting from *start address*

This will give an error if:
The target computer is not running the clever download software
There is a fault in the link to the target computer

NOTE:

This will show a block of memory on the display, showing the hex and ascii. You may move the cursor up the screen and alter the hex bytes, pressing enter on the line will then download it, changing the bytes altered. The line will then be uploaded again and reprinted. This is so if you were to alter ROM, it would not change when you pressed enter on the line.

N - Send a command down to the download software

N *byte value* [, *byte value* , *byte value* , *byte value* ....]

byte value
Is a value that is sent to the target computer

This command is included to allow you to expand the uses of the PDS monitor yourself. It will send this byte down to the download software. The download software protocols in PDS use, or will use the bytes in the range 170 - 200 to specify certain functions, so try and avoid these unless you have changed the target computers download software. The current version of the DL1 software only uses bytes in the range 179 to 187 to specify commands, although the others will be used for future expansion. This command is very powerful, although you will have to write the software yourself. Basically, add a say, 'CMP #5, BEQ MINE' to the main loop of the target computer download software, then in your MINE routine do something clever. To initiate this clever routine from the monitor, just type N5. You could for instance have N0 to freeze the program when used with DL2, under the interrupts software, and N1 to restart the program again. Or you could have N0 to exchange screens in a two screen game. Remember that multiple bytes can be sent, so that complicated commands can be sent to the target computer. You will need to become familiar with the download-software to use this command fully.

O - Set an offset for all addresses

There are two ways to use this command:

1] O

This will show the current offset being added to all addresses

2] O *offset*

This will set the new offset to *offset*

All memory addresses sent to and from the target computer will have this *offset* added to them first.

For example, if I had a program loaded in the target computer at $8000 and it was supposed to be at $9000h, then if I entered:

```
O $8000-$9000
```

From then on it would be as if the program really was at $9000, for example D $9000 would disassemble the start of the program.

P - Work out screen coordinates or addresses

There are three ways to use this command:

1] P *address*

This will return the x and y coordinates of the screen *address*.

2] P *x,y*

This will return the screen address of the *x* and *y* coordinates.

3] P@ *address*

This will set the start of the screen to *address*

NOTE:

These commands do not require a target computer connected. The computer must be setup in the config system or using the 'Q' command before this command is used. There are six different screen formats supported by this command:

0. ZX-SPECTRUM - Hires screen at $4000, 256 by 192 pixels
1. COMMODORE 64 - Hires screen at 0, 300 by 200 pixels
2. AMSTRAD MODE 0 - Hires screen at $C000, 160 by 200 pixels
3. AMSTRAD MODE 1 - Hires screen at $C000, 320 by 200 pixels
4. AMSTRAD MODE 2 - Hires screen at $C000, 640 by 200 pixels
5. SPECIAL AMSTRAD - Mimic spectrum mode' at $C000, 256 by 192

If you are using a second screen , or you have moved the screen from the defaults listed above, use the P@ *address* command to specify the new start address.

When you use the P *address* command, the coordinates return will be for the left most pixel in that byte.

EXAMPLES:

(All these assume that the current computer is 1, Commodore 64)

P0,0	Will print :	Addr = $0000, Bit = $80
P103,48	WIll print :	Addr = $07E0, Bit = 01h
P$1234	Will print :	X = 176 , Y = 116
P$5143	Will print :	That address is not on the screen!

You will notice how a commodore bitmap address range is from 0 to $3FFF, this address has to be added to the current screen base address, before you will get the correct screen address.

Q - Read or set options

This command can be used in two ways:

1] Q *option*

Will print the current value of the *option*.

2] Q *option* , *value*

Will set the *option* to the *value* given.

NOTE:

This command does not require a target computer connected. All the options can also be found in the config system, see the editor manual for more details. For a list of all the options and their parameters see the chapter on options.

EXAMPLES:

The sequence of commands below will take a screen address on the Spectrum and return the equivalent address on the Amstrad screen.

```
Q1,0          - Select Spectrum as default computer
P$4567        - Find the coordinates of a screen address
Q1,1          - Select Amstrad as default computer
P56,29        - Find the screen address given coordinates
```

The sequence of commands below will take an area of memory from the Spectrum and transfer it to a Commodore 64. The Spectrum is connected in port 1, the Commodore 64 in port 2.

```
Q0,1          - Select port 1 (The Spectrum)
U>DEMO,$4000,1000  - Upload 1000 bytes from $4000
Q0,2          - Select port 2 (The Commodore 64)
U<DEMO,$C000  - Download the bytes to address $C000
```

Instead of entering values using the assembler expression evaluator, you may enter them in hex only, for example:

```
M$A987        - Will modify memory address $A987
Q3,1          - Set 'HEX expressions'
MA987         - Now all numbers are treated as hex
```

R - Set or update the register values

The two formats for this command are:

1] R

This will update all the registers at the top of the screen

2] R *register* = *value*

This will set the *register* to the *value* specified

register
Is any 6502 register, for example A, X, Y, or PC (16 bit).

This will given an error if:

The clever download software is not being used.
There is a fault in the link to the target computer.
The *value* is too big for the *register* specified.

NOTE:

You may set more than one register at a time, for example:

```
R X=23,A=2,PC=$4000
```

The registers at the top of the screen are only updated after tracing has finished, or an R command has been executed. If you wish to update them yourself, enter R on its own. PDS could not keep them updated all the time because you may not have a target computer connected, or it may not be running clever download software.

S - Search memory for data

S *start address* , *end address* , *data*

start address
Is the first byte that will be scanned in the search

end address
Is the last byte that will be examined

data
Is the data the is being searched for

This will give an error if:
The clever download software is not being used.
There is a fault in the link to the target computer.
The *end address* is lower than the *start address*.

NOTE:

The data may be any number of bytes, for example:

S0,100,23	- Will search the first 100 bytes for 23's
S0,100,1,2,3	- Will search for the pattern 1,2,3
S0,100,"PDS"	- Will search for the string 'PDS'

PDS will search the memory by uploading the block and searching through it in the main computers memory. If it finds a match then it will print the address of the first byte of the match. It will the carry on the search from the byte following this address.

T - Trace

This command takes two formats:

1] T

This start tracing from the current PC

2] T *address*

This will start tracing from *address*

The PDS tracer is a symbolic tracer. This means that if a program had just been assembled, the tracer will use them while it traces. To use symbolic tracing, ensure that the 'Q5' option has been correctly setup (see chapter 6.3 and the 'Q' command in the monitor for more details).

This will give an error if:
The clever download software is not being used.
There is a fault in the link to the target computer.

NOTE:

This is one of the most complex features in the whole of the development system. For full information see the chapter on tracing.

U - Upload/Download memory from/to the target computer.

This command has three different formats:

1] U *start address* , *length* [,*bytes per line*]

This will upload an area of memory and place it at the current editor cursor position in the form of HEX statements, each line will have 16 bytes, if *bytes per line* is not specified.

2] U >*filename* , *start address* , *length*

This will upload an area of memory from the target computer into a disk file called *filename*. Note the U

3] U <*filename* , *start address* , *length*

This will download a disk file into the target computer at the start address, if length is not specified then the whole file is transferred. Note the U

This will give an error if:
The clever download software is not being used.
There is a fault in the link to the target computer.
3] *length* is longer than the file on the disk.
3] *filename* does not exist.
1] Too much memory is uploaded, not enough space in the file.

NOTE:

The file name in versions 2] and 3] may contain drives and path names. In version 1] you may have 1 to 55 bytes per line in the HEX statements.

EXAMPLES:

To upload the default screen from a Commodore 64 into a file in the current directory called SCREEN.C64 use:

```
U>SCREEN.C64,$400,1024
```

To download it again to the commodore screen use:

```
U<SCREEN.C64,$400
```

V - Show nearest label

V *expression*

expression
This can be any number or normal expression.

Once entered, this will display the nearest label before, after and at that address. A message will be displayed if no match was found. This is very useful when you are trying to trace or disassemble your code, as you can then reference addresses with labels in your code. Make sure that you assemble your program before hand, as a symbol table is required if this function is to work.

EXAMPLE:

If BILL = $3E8, TOM = $514, and DICK = $11B4. Then entering the following command :

```
V2000
```

Will return:

```
TOM ($0514), 02BC bytes below that address
No labels at that address
DICK ($11B4), 09E4 bytes above that address
```

W - Change or display a word at a memory location

This command can be used in two ways:

1] W *address*

This will display the word at the *address* given. The value will be displayed in hex, decimal, binary and ascii.

2] W *address* , *value*

This will move the *value* into the address given.

address
This is the address of the word to be changed

value
This is a value, from -32768 to 65535

This will give an error if:
There is a fault in the link to the other computer
1] Needs the clever PDS download software
2] Only needs the dumb download software

NOTES:

The B command has the same function, but works on bytes instead of words. This command is mainly used for altering values in your program, to quickly see the effect of small changes.

EXAMPLES:

```
W$4000,23
```

This will load the location $4000 with the value 23, and location $4001 with 0

```
W FRED,{FRED}*2
```

This will double the word value at locations FRED and FRED + 1

X - Evaluate an expression

X *expression*

expression
Is a valid expression that will be evaluated using the assemblers expression evaluator. If you have just assembled something then you may use label names.

NOTE:

This does not require a target computer attached.
The value of the *expression* will be printed in hex, decimal, binary and if within the range 32 to 127, in ascii.

EXAMPLE:

```
X23+45*2
HEX=0071 , DEC=00113 , BIN=00000000-01110001 , "q"
```

Z - Change the bank selected in the target computer

Z *bank number*

bank number is a value sent down to the target computer.

This will give an error if:
There is a fault in the link to the target computer

This command varies on all computers, in the download software you will find a small section you have to write yourself. This is a section that gets the *bank number* 0-255 and sets up the computers memory depending on that value. If you wish you could 'OUT' it straight to a 'bank select' port, on a spectrum 128 for instance. Or you could have a more complex routine, with different setups for each value of BANK. This command has been written like this as all banked ram computers are different and most programmers have different systems, or values for each different bank configuration.

6.3 The options and configure system in the monitor

The monitor can be changed or altered by a number of options, which can be set or examined either in the config system from the editor, or by using the Q command in the monitor. Below are all the Q options and what they can change.

Option	Range	Alters
0	1 or 2	Which PDS port the monitor works from, the default is port 1.
1	0 to 5	Which computer PDS thinks it is attached to, this is only really used in the P command so that PDS knows how the screen is addressed.
2	0,2-16	This sets the radix of all the monitor commands, the default is 10. If it was set to 2 then D1010 would disassemble from location 10 (decimal) onwards. If this is set to 0 then the commands only accept hex numbers. No expressions are allowed, you could then use commands such as DABCD or MC9.
3	0-255	This sets the number of lines displayed in the M command. The default is 8. If it is set to 0 then memory will displayed one screen at a time, and key will move onto the next screen, the up arrow will move backwards one screen. Escape or finish will quit this command.
4	0-255	This sets the number of lines displayed in the D command. The default is 8. If it is set to 0 then code will be disassembled one screen at a time, any key will go onto the next screen. Finish or escape will quit this command.
5		Disassembler flags, each bit in this byte controls a function in the disassembler.
	128 (7)	1 = All expressions in decimal, else in Hex.
	64 (6)	1 = Expressions in ascii if in range.
	32 (5)	1 = Tab after the instruction, else a space.
	16 (4)	1 = Initial address in decimal, else in Hex.
	8 (3)	1 = IX + nn and IY + nn always in decimal, else current radix is used.
	4 (2)	1 = Labels will be printed at the beginning of the line during disassembly or tracing.
	2 (1)	1 = Labels will be printed with the instruction being disassembled or traced.
	1 (0)	1 = If an exact label match is not found, then offsets to that label will be displayed. The offset can vary from -2 to +2.
6		Trace system flags. Each bit controls a function :
	128 (7)	1 = When executing in 'FAST' mode, do a VSYNC after

each instruction, This makes the bar visible all the time on the screen, although it slows the tracer down to 50 instructions a second.

7

General monitor commands flags

128 (7) 1 = The E command accepts decimal input, else hex.

64 (6) 1 = Analyze doesn't stop when one bar reaches the end, it will keep going until a key is pressed.

8

Sprite grabber flags

8 (3) 1 = The cursor will move 8 pixels at a time in the Y direction.

4 (2) 1 = The cusor will be stored on the screen, otherwise it is xor'd onto the screen.

2 (1) Cursor type, high bit

1 (0) Cursor type, low bit

00 - The default 8 by 1 pixel, single byte cursor is used.

01 - The four corner bytes of the sprite are used as the cursor.

10 - The top and bottom pixel line of the sprite is used as the cursor.

11 - The cusor becomes an inverted block, the size of the sprite

6.4 The trace system

You can start to trace a program by entering T address as a command in the monitor. If the PC displayed at the top of the screen is already pointing at the start of your program, you may press just T to start tracing.

You will need the clever download software in the target computer to trace successfully. When you enter the tracer you will see 15 lines of source code disassembled along with their addresses and hex. There will be an inverted bar on the first line of this disassembly, this shows the current instruction. At the bottom of the screen there is a small box, this is for memory displays. Above this there are all the options, with their first letters highlighted.

When PDS traces a program it does not really execute the code, if it did then when it executed a jump for instance it would exit the PDS download software and run your program at full speed. It manages to trace programs by half running them and half emulating them. When PDS comes across say, JMP $1234 it would just load PC with $1234. For more complex instructions such as ADC #2 it downloads the instruction along with a couple of padding NOPS into a three byte buffer in the clever download software. It restores all the registers, executes the instruction, saves all the registers, sends them back up to the main computer and then carries on with the next instruction. It has to do this with most instructions it executes to get the flags altered correctly. Even doing all this work, on an IBM PC it can execute about one thousand instructions a second.

If PDS comes across an unknown instruction, it will place the inverted bar on top of it and beep, it will not continue until the value of the PC is altered to either another part of your program, or just past that instruction. You could put illegal instructions around your program to halt the tracer, $FD for instance.

The tracer can use the sybols from your program, while it is tracing. To do this, ensure that your program had just been assembled, and the option 'Q5' is setup accordingly (see chapter 6.3).

Note that the programs you trace should not access the ports used by the PDS interface, this will almost certainly stop the tracer, and probably crash the target computer. You should also notice that while the clever download software is used, the interrupts are always disabled. The tracer will show when interrupts are enabled or disabled, although they never really alter. If you trace a HALT, it will be treated just like a NOP by the tracer (unless the interrupts are disabled). If the interrupt routine is important (say reading keyboards) then call them now and again from the tracer.

6.5 The tracer options

There are 11 different options you may choose in the tracer main menu they are :

Any key
Will execute the next instruction and show and registers changed.

G - 'Go full speed'
Will execute a jump into your program at the current PC. This will leave the download software and run your program full speed.

F - 'Fast trace'
This will execute code fast, as if you kept a key held down. It will stop when you press any key.

Q - 'Quick trace'
This will execute code quickly, it will only update the PC in the register display, it will not disassemble the current instruction on the screen and will not update any other register values. This speeds up the tracer a lot and is used to keep an eye on where it is without slowing it down too much. It will stop when you press any key.

S - 'Stack display'
This will open or close a box on the right hand side of the disassembly. This box shows the contents of the last ten items pushed on the stack. When a PHA is executed you will see the value of A and the letter A on the bottom of the stack. If you execute a JSR, then '$L' and '$H' are displayed at the relavent stack position. This is so you can see if the correct registers are being pulled off the stack in order. To turn this stack off, press S again.

R - 'Register values'
Allows you to change register values, it will prompt you for the register and value in the form *reg* = value. *value* can be any expression as long as it is within range. < reg > can be any eight or sixteen bit register.

M - 'Memory display'
This will allow you to see a block of 32 bytes in hex and ascii, the block can be at a fixed address, or pointed to by a register. To switch off the memory display, select M again. The memory display is updated after every instruction.

L - 'Execution limits'
Allows you to specify a range of addresses that can be traced, if the program goes out of these limits then it will stop and beep. The upper and lower limits are default set to 0 and $FFFF.

N - 'Execute the next instruction'
The tracer will execute the next instruction at full speed. This is mainly used for executing routines at full speed. Say the tracer was about to execute JSR CLS, you don't really want to trace this routine so just press N. If the routine is very long and takes over a second to execute then the tracer may think the target computer has hung. If this occurs you will just have to wait until the routine has finished and the enter T for trace continue.

I - 'Ignore this instruction'
This will make the tracer skip over the current instruction. It is mainly used for skipping out of loops, or avoiding calling routines.

T - 'Trace very quickly until condition'
Will trace as fast as it can, with no display updating, until a condition is met. Once T is pressed, a second menu is displayed containing the conditions.

Key - 'K'
Will trace until any key is pressed. This is the fastest trace available.

Break - 'B'
Will trace until a BRK is executed (many programs use this for message or error trap routines).

Return - 'R'
Will trace until an RTS is executed.

Stack - 'S'
Will trace until the stack is altered in any way.

Fail - 'F'
Will trace until a conditional instruction fails, or does not jump.

Pass - 'P'
Will trace until a certain point is passed a certain number of times. This can be used as a breakpoint. It can also display the number of times the point has been passed as a countdown.

Compare - 'C'
Will trace until a register or memory address either goes outside set limits or until a particular bit pattern occurs.