

**Reverse Engineering of the (guessed) operation of the 6702 chip to enable use of the VICE emulator in 6809 mode with the Waterloo Language Software and removal of the protection from the Waterloo Language Disks to enable users of a real SuperPET (but with a faulty 6702) to continue to enjoy their machine.**

**Background**

The Commodore SuperPET (also known as the SP9000) contains two microprocessor sub-systems: a Mostek 6502 and a Motorola 6809. A common memory and peripheral sub-system is shared between the two microprocessors. Because the two microprocessors utilise different instruction sets, two sets of ROMS are required – one set for the 6502 and the other set for the 6809. An external switch allows the user to select which of the two microprocessors take control of the machine at any given time.

The 6502 ROMS make the system behave in a manner similar to a ‘normal’ 80 character-wide screen PET. When the SuperPET is switched into 6809 mode – the ROMS only contain a basic ‘bare’ operating system (enough to load software from an attached floppy disk unit). The disk-resident software languages (microAPL, microBASIC, microFORTRAN, microPASCAL, microCOBOL and a 6809 Assembler) were developed by the University of Waterloo in Ontario, Canada.

The SuperPET contains an integrated circuit identified as a “6702”. No known information is available on this device. It has been assumed that this 6702 device is an early form of software protection as the Waterloo software fails to operate correctly should this device fail or be removed (early SuperPETs had the 6702 integrated circuit mounted on a small daughter board).

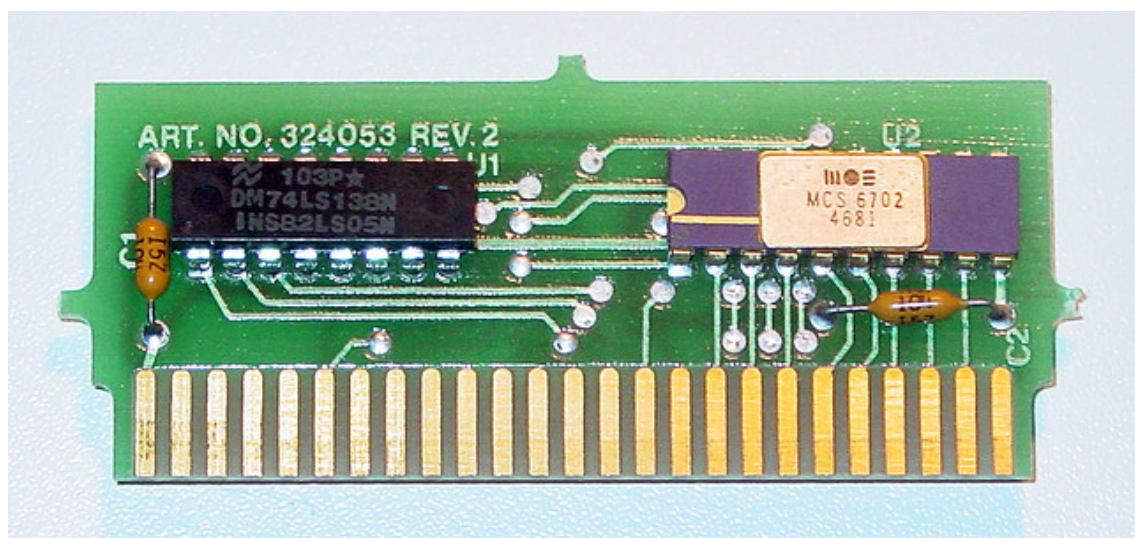


Figure 1 - The daughter board with the 6702 device (Photograph by Mike Naberezny).

VICE (the VersItile Commodore Emulator) is a software project to enable Commodore software to execute on more modern hardware and software platforms (e.g. a modern PC). Currently, VICE only supports the Commodore range of products that utilise the 6502 microprocessor. This includes the SuperPET – but only in 6502 microprocessor mode. Developers have been working to incorporate a 6809 microprocessor emulator within VICE with the intention of enabling the use of the Waterloo Language disks.

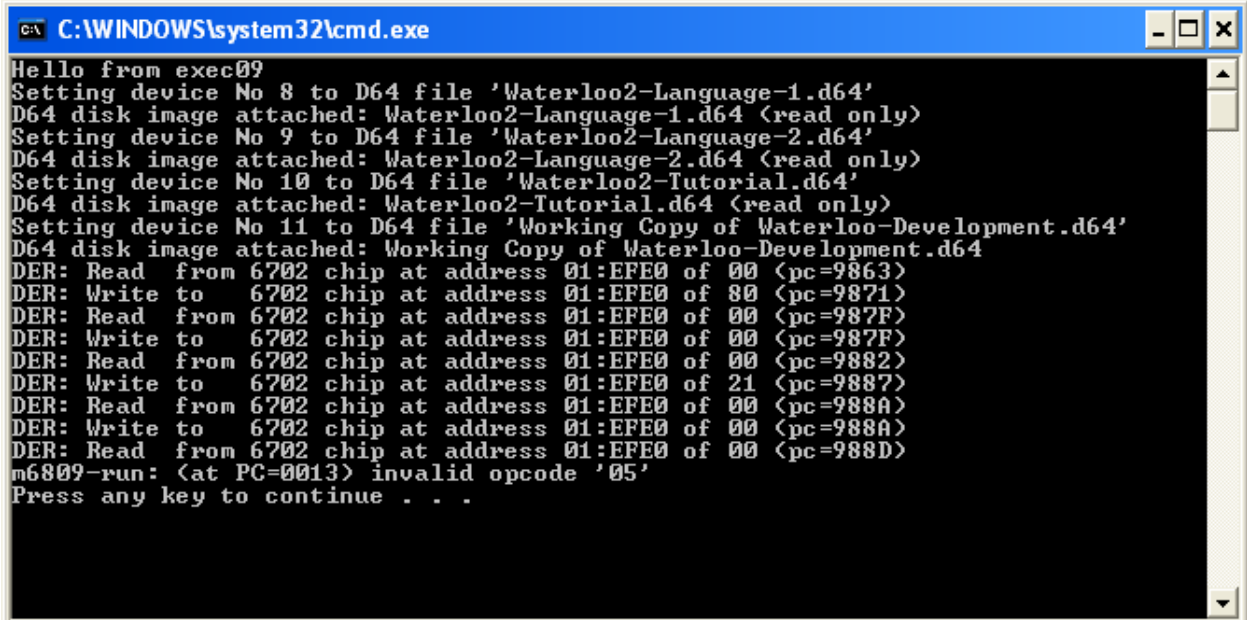
Once the VICE 6809 emulator was accurate enough to execute the resident ROM software, problems immediately became evident as soon as the first software product was loaded from disk and executed. The software promptly addressed the non-existent 6702 device within the emulator and crashed. This was not unexpected behaviour!

The problem now was to identify what the 6702 integrated circuit actually did without any prior knowledge of the device's operation.

Please note that this document does **not** condone the illegal copying and use of unlicensed software. This text has been written to enable legitimate owners and users of SuperPET software to continue to use and enjoy their computer even if the 6702 device becomes defective and cannot be replaced.

### Sleuthing

A 'dummy' 6702 function was created within the software emulator to trap and display all attempts by the executing 6809 software to access the peripheral addresses occupied by the 6702 device. This enabled the bank number and the memory address of the instruction causing the memory address to be displayed (along with the data it was writing if it was a write access). A read access would always return a fixed, pre-determined number (0 in my case).



```
C:\WINDOWS\system32\cmd.exe
Hello from exec09
Setting device No 8 to D64 file 'Waterloo2-Language-1.d64'
D64 disk image attached: Waterloo2-Language-1.d64 (read only)
Setting device No 9 to D64 file 'Waterloo2-Language-2.d64'
D64 disk image attached: Waterloo2-Language-2.d64 (read only)
Setting device No 10 to D64 file 'Waterloo2-Tutorial.d64'
D64 disk image attached: Waterloo2-Tutorial.d64 (read only)
Setting device No 11 to D64 file 'Working Copy of Waterloo-Development.d64'
D64 disk image attached: Working Copy of Waterloo-Development.d64
DER: Read from 6702 chip at address 01:EFE0 of 00 (pc=9863)
DER: Write to 6702 chip at address 01:EFE0 of 80 (pc=9871)
DER: Read from 6702 chip at address 01:EFE0 of 00 (pc=987F)
DER: Write to 6702 chip at address 01:EFE0 of 00 (pc=987F)
DER: Read from 6702 chip at address 01:EFE0 of 00 (pc=9882)
DER: Write to 6702 chip at address 01:EFE0 of 21 (pc=9887)
DER: Read from 6702 chip at address 01:EFE0 of 00 (pc=988A)
DER: Write to 6702 chip at address 01:EFE0 of 00 (pc=988A)
DER: Read from 6702 chip at address 01:EFE0 of 00 (pc=988D)
m6809-run: (at PC=0013) invalid opcode '05'
Press any key to continue . . .
```

Figure 2 - Sample session with a 'dummy' 6502 function.

Notice the ‘crash’ at the end as the Waterloo editor detects a non-responsive or faulty 6702 and forces a subroutine to be executed within the ROM called ‘suicide’ – you can work out for yourself what that subroutine does from its name!

From the list of addresses, it was then possible to use a 6809 debugger to disassemble the subroutine code used to access the 6702.

A screenshot of a window titled "SP9000" with a blue title bar. The window contains a terminal window titled "Waterloo microMonitor" with a black background and green text. The text shows a sequence of commands and their output: ">1 EDIT", ">b 1", ">t 98a4-98bd", followed by a list of assembly instructions with their addresses: 98a4 ANDA ,Y; 98a6 PSHS #02; 98a8 LDA ,X+; 98aa ANDA ,Y+; 98ac SUBA ,S+; 98ae BNE \$98ba; 98b0 ASL #01,S; 98b2 BNE \$98a2; 98b4 LDX #02,S; 98b6 LSRB; 98b7 LSRB; 98b8 BNE \$986b; 98ba LEAS #0e,S; 98bc RTS. The session ends with ">■".

```
SP9000
Waterloo microMonitor
>1 EDIT
>b 1
>t 98a4-98bd
98a4 ANDA ,Y
98a6 PSHS #02
98a8 LDA ,X+
98aa ANDA ,Y+
98ac SUBA ,S+
98ae BNE $98ba
98b0 ASL #01,S
98b2 BNE $98a2
98b4 LDX #02,S
98b6 LSRB
98b7 LSRB
98b8 BNE $986b
98ba LEAS #0e,S
98bc RTS
>■
```

Figure 3 - Sample debug session on the Waterloo EDITor.

For those that are interested, there follows a disassembly of the 6702 validation subroutine from bank number 1 of the Waterloo microEditor starting from memory address \$9852. Notice that the above disassembly (obtained from the Waterloo microMonitor) does not exactly match the code below. The disassembly below was obtained from the VICE 6809 debugger – which decodes the instruction byte sequence slightly more correctly than the Waterloo microMonitor (see the instruction at address \$98A6 for an example).

```

9852 1F 41      TFR S, X
9854 32 72      LEAS -14, S
9856 86 F0      LDA #F0
9858 A7 66      STA 6, S
985A 6A 66      DEC 6, S
985C A7 67      STA 7, S
985E 68 67      ASL 7, S
9860 A6 F8 06   LDA [$06, S]      ; 6702 DEVICE READ
9863 8A 11      ORA #S11
9865 A7 64      STA 4, S
9867 A7 65      STA 5, S
9869 C6 40      LDB #S40
986B 58         ASLB
986C E7 61      STB 1, S
986E E7 F8 06   STB [$06, S]      ; 6702 DEVICE WRITE
9871 EC 64      LDD 4, S
9873 3D         MUL
9874 E7 65      STB 5, S
9876 AA 65      ORA 5, S
9878 A7 64      STA 4, S
987A C8 D7      EORB #SD7
987C 68 F8 06   ASL [$06, S]      ; 6702 DEVICE READ and WRITE
987E E8 F8 06   EORB [$06, S]     ; 6702 DEVICE READ
9882 A6 65      LDA 5, S
9884 A7 F8 06   STA [$06, S]      ; 6702 DEVICE WRITE
9887 68 F8 06   ASL [$06, S]      ; 6702 DEVICE READ and WRITE
988A A6 F8 06   LDA [$06, S]      ; 6702 DEVICE READ
988D A7 E4      STA , S
988F E7 82      STB , -X
9891 AF 62      STX 2, S
9893 E6 61      LDB 1, S
9895 8D 08      BSR $989F
9897 10 80      FCB $10, $80
9899 22 00      FCB $22, $00
989B 40         FCB $40
989C 00 04      FCB $00, $04
989E A8         FCB $A8
989F 10 AE E1   LDY , S++
98A2 AA E4      ORA , S
98A4 A4 A4      ANDA , Y
98A6 34 02      PSHS A
98A8 A6 80      LDA , X+
98AA A4 A0      ANDA , Y+
98AC A0 E0      SUBA , S+
98AE 26 0A      BNE $98BA
98B0 68 61      ASL 1, S
98B2 26 EE      BNE $98A2
98B4 AE 62      LDX 2, S X
98B6 54         LSRB
98B7 54         LSRB
98B8 26 B1      BNE $986B
98BA 32 6E      LEAS 14, S
98BC 39         RTS

```

Figure 4 - Sample disassembly of the Waterloo microEditor 6702 validation subroutine.

By running each of the Waterloo language packages in turn it was noted that each of the software subroutines that addressed the 6702 device appeared to be identical – albeit starting at a different bank number and/or memory address. Disassembly proved that the code within the subroutines were, in fact, identical.

### Attempt #1

The first attempt to disable the 6702 checking consisted of identifying the start address of the validation subroutine in each of the Waterloo languages, determining the value in the 6809 processor registers that should have been returned to the caller if a successful 6702 byte

sequence had been detected, and patching the code to return these values. This solution failed to work! It was later identified that the 6702 subroutine code is 'check summed' by another embedded subroutine elsewhere. Any attempt to patch the 6702 validation subroutine without making a corresponding change to the checksum algorithm is doomed to failure. I know this now!

### Attempt #2

It was identified that all accesses to the 6702 validation subroutine went through the ROM code 'bankswi' (bank switch) subroutine. This subroutine maps in the correct bank of memory into the address window from \$9000 through \$9FFF and performs a JSR (Jump to Sub-Routine) to the specified address in the bank switched memory. A return from the called subroutine then re-enters the ROM 'bankswi' subroutine once again which restores the bank of memory in force prior to the call and returns back to the original caller.

The 6809 emulator's JSR instruction was then modified to detect the bank number and target address for the call and to \*\*\* **NOT** \*\*\* invoke the subroutine if it matched a table of specific addresses (previously identified in #1 above). Under this circumstance, the 6809 processor's A and B registers and flags would be set to the correct state to 'fake' a 'good' return from the 6702 validation subroutine without affecting the actual byte sequence of the subroutine itself in memory. This prevents the (currently unidentified) checksum subroutine from detecting an intrusion!

This solution almost worked! Further investigation of the ROM 'banksw' subroutine identified that it was also possible for this code to determine that the caller and the target subroutine were in the same memory bank as each other and that no bank switch was therefore necessary. Under these circumstances a JMP (JuMP) was performed rather than a JSR. This necessitated that a similar modification was undertaken to the 6809 emulator's JMP instruction.

It was considered that this modification to the 6809 emulator could result in 'false positives' being identified (i.e. legitimate code from one language calling a valid subroutine in the same bank and at the same address as a completely different language). This 'false positive' could have resulted in an incorrectly functioning language processor as the target subroutine would not have been executed, and the 6809 processor registers potentially corrupted! To overcome this potential pitfall, the 6809 emulator was further modified to contain additional checks thus minimising the probability of this occurring (e.g. that the JSR or JMP instruction was occurring at a known address within the ROM 'bankswi' subroutine and that the target address contains the correct first few bytes of the expected 6702 validation subroutine).

The bank numbers and addresses for the **second release** of the Waterloo Language software are given in the table below:

Waterloo Language	Software Version	Date	Bank Number	Subroutine Address (Hex)
EDIT	1.1	1982	1	9852
PASCAL	1.1	1982	5	9000
BASIC	1.1	1981	5	93F0
FORTRAN	1.1	1981	0	960C
APL	1.1	1982	6	9F12
COBOL	1.0	1981	0	9240
DEVELOPMENT ASSEMBLER	1.1	1981	5	9A05
DEVELOPMENT LINKER	1.0	1981	8	94B6
DEVELOPMENT EDITOR	1.1	1982	2	9852

**Figure 5 - Table of 6702 validation subroutine addresses for the various Waterloo languages.**

This resulted in a correctly functioning suite of Waterloo Language software – but at the expense of hard-coding the 6809 emulator with a list of addresses specific to the software being executed. These modifications were only valid for the second release of the Waterloo Language Disks. The first release of the Waterloo Language disks contained different addresses (as the software build is slightly different) and the solution would not therefore work.

The 6809 emulator was subsequently updated to include the addresses for the first release – but at the expense of further hand-crafting!

Whilst this solution works in a software emulator environment (like VICE) – it is neither elegant nor will it help existing SuperPET owners where their 6702 integrated circuit has become defective.

### **Attempt #3**

By analysing a disassembly of the 6702 validation code (determined from the above table of bank numbers and addresses), the following course of action was identified:

- Ignore all data writes to the 6702 and concentrate on the data reads from the device only.
- Simplify the 6702 validation subroutine as far as possible by removal of what appeared to be superfluous code. This identified that the subroutine accessed 15 data bytes in total from the 6702 to perform one complete successful pass of the 6702 validation algorithm.
- Code the resulting simplified algorithm up in 'C' and use a random number generator (a Monte Carlo method) to 'guess' at the 15 data bytes that may have been returned

from the 6702. Run the validation algorithm to identify if the guessed solution is the correct one or not. Keep repeating until a successful guess has been identified and print the result out. Keep iterating to find any further valid solutions.

This course of action identified a number of 15-byte sequences that would cause the existing 6702 validation code to 'think' that it was communicating with a valid 6702!

```
Hit 1: 04 91 3C AC C7 F0 E3 36 5A BB C3 6A 75 A3 3E
Hit 2: 67 96 9C 53 4B A4 08 F0 E3 5A 0A B7 87 5F 8B
Hit 3: 59 0F 8B 8C 35 17 3D 48 18 A5 A0 F0 C0 8C 0A
Hit 4: D5 A4 23 75 5E 31 8F 5A 02 D5 4F 0E 26 45 93
Hit 5: D6 4C 6D 17 27 2D 5E 56 D0 31 AD 2D 60 06 52
Hit 6: 03 3D 83 2C AF D8 CE 05 E3 53 29 64 EA 6F 5A
Hit 7: 91 58 85 2A A1 51 27 82 2E FC 0F 83 A2 5B 6E
```

**Figure 6 - Sample of successful Monte Carlo hits for the 6702 data table values.**

The hand-crafted hacks to the JMP and JSR instructions of the 6809 emulator were immediately removed and code inserted into the 'dummy' function for the 6702 emulator to return one of the previously identified byte sequences (the first hit if my memory serves me correctly). This necessitated expanding the 15-byte data table to include additional bytes used to return 'dummy' values to 6809 instructions that did not appear to have any bearing on the logic (e.g. the ASL instruction where a value read from the 6702 was modified by the 6809 instruction and fed back to the 6702 – but the data actually read from or written to the 6702 was actually discarded by the validation subroutine).

All instruction data writes to the dummy 6702 function are ignored by the emulator (e.g. the STA and STB instructions identified in the disassembly).



```

case 0xEFE0 :|
case 0xEFE1 :
case 0xEFE2 :
case 0xEFE3 : { static const tt_u8 CODE_TABLE[ 29 ] = {

    // Used once at the start of the subroutine.
    0x04, // 9860: LDA [<$06,S] ; [ 0]

    // Iteration 1 of the 'outer' loop.
    0xFF, // 987C: ASL [<$06,S] ; Can ignore!
    0x91, // 987F: EORB [<$06,S] ; [ 2]
    0xFF, // 9887: ASL [<$06,S] ; Can ignore!
    0x3C, // 988A: LDA [<$06,S] ; [ 4]

    // Iteration 2 of the 'outer' loop.
    0xFF, // 987C: ASL [<$06,S] ; Can ignore!
    0xAC, // 987F: EORB [<$06,S] ; [ 6]
    0xFF, // 9887: ASL [<$06,S] ; Can ignore!
    0xC7, // 988A: LDA [<$06,S] ; [ 8]

    // Iteration 3 of the 'outer' loop.
    0xFF, // 987C: ASL [<$06,S] ; Can ignore!
    0xF0, // 987F: EORB [<$06,S] ; [10]
    0xFF, // 9887: ASL [<$06,S] ; Can ignore!
    0xE3, // 988A: LDA [<$06,S] ; [12]

    // Iteration 4 of the 'outer' loop.
    0xFF, // 987C: ASL [<$06,S] ; Can ignore!
    0x36, // 987F: EORB [<$06,S] ; [14]
    0xFF, // 9887: ASL [<$06,S] ; Can ignore!
    0x5A, // 988A: LDA [<$06,S] ; [16]

    // Iteration 5 of the 'outer' loop.
    0xFF, // 987C: ASL [<$06,S] ; Can ignore!
    0xBB, // 987F: EORB [<$06,S] ; [18]
    0xFF, // 9887: ASL [<$06,S] ; Can ignore!
    0xC3, // 988A: LDA [<$06,S] ; [20]

    // Iteration 6 of the 'outer' loop.
    0xFF, // 987C: ASL [<$06,S] ; Can ignore!
    0x6A, // 987F: EORB [<$06,S] ; [22]
    0xFF, // 9887: ASL [<$06,S] ; Can ignore!
    0x75, // 988A: LDA [<$06,S] ; [24]

    // Iteration 7 of the 'outer' loop.
    0xFF, // 987C: ASL [<$06,S] ; Can ignore!
    0xA3, // 987F: EORB [<$06,S] ; [26]
    0xFF, // 9887: ASL [<$06,S] ; Can ignore!
    0x3E, // 988A: LDA [<$06,S] ; [28]

};

```

Figure 7 - Snippet of code from the 6809 emulator's dummy 6702 function showing the lookup table.

This also resulted in a fully working 6809/6702 emulator without any hand-crafted internal hacks.

This solution could be extended into hardware by implementing the look-up table in an EPROM and arranging for a counter to cycle through the look-up table for each read from the



6702 device. Writes to the 6702 device should be ignored. The counter should be reset to zero on power-up.

Postscript: The first analysis of a real 6702 indicates that it does not appear to work this way and that the writes to the 6702 do (in fact) alter the returned read values. To be continued...

#### Attempt #4

Interest has been shown in removing the 6702 protection totally from the Waterloo Language disks. Some success has previously been obtained with the microEditor – but this has not been extended to the other Waterloo language tools.

As we already know the address of the start of the 6702 validation subroutine for each language module it was possible to patch the subroutine with instruction bytes to return with the expected result in the 6809 CPU registers. It was also already known that this solution did not work as expected due to the presence of a checksum subroutine. The checksum subroutine was hunted down in the microEditor and the checksum of the existing 6702 validation subroutine determined by hand by inspection of the disassembly. The checksum of the required patches to the 6702 validation subroutine was then determined and the modification to the checksum subroutine identified to make the resulting check pass.

```

9B8E 32 7D      LEAS -3,S
9B90 6F 62      CLR 2,S
9B92 CC 98 52   LDD #$9852
9B95 ED E4     STD ,S
9B97 AE E4     LDX ,S
9B99 E6 84     LDB ,X
9B9B C1 39     CMPB #$39
9B9D 27 10     BEQ $9BAF
9B9F 4F        CLRA
9BA0 E6 62     LDB 2,S
9BA2 EB 84     ADDB ,X
9BA4 89 00     ADCA #$00
9BA6 E7 62     STB 2,S
9BA8 EC E4     LDD ,S
9BAA C3 00 01  ADDD #$0001
9BAD 20 E6     BRA $9B95
9BAF 4F        CLRA
9BB0 E6 62     LDB 2,S
9BB2 C3 FF 03  ADDD #$FF03
9BB5 E7 62     STB 2,S
9BB7 4F        CLRA
9BB8 E6 62     LDB 2,S
9BBA 32 63     LEAS 3,S
9BBC 39        RTS

```

Figure 8 - Sample disassembly of the Waterloo microEditor checksum subroutine.

The astute reader will identify that the operand address located at instruction \$9B92 of this disassembly (\$9852) matches with the start address of the microEditor 6702 validation subroutine disassembly presented in Figure 4; and that the immediate operand value located at instruction \$9B9B of this disassembly (\$39) matches with the hexadecimal coding of the RTS instruction at the end of the disassembly of the subroutine presented in Figure 4.

Fortunately, the code for both the 6702 validation subroutine and the checksum subroutine appears to be identical across all of the Waterloo Language software. The initial byte sequences of interest were identified in the D64 disk images and checked further by hand to ensure that they corresponded to the actual subroutines of interest. A check was also made to ensure that the identified checksum subroutine actually pointed to the address of an already identified 6702 validation subroutine!

Each of the identified disk addresses were converted into track and sector numbers and byte offsets within the sector.

The d64Editor.exe utility was then used to first verify that the identified track and sector combinations did actually belong to the expected utility (another cross-check) and to perform the patch.

Each patch was performed in a logical manner:

- Make sure that the Waterloo Language tool worked as expected with the 6809 emulator without the patches. Ensure that the emulator reported 6702 accesses.
- Make the patch to the selected 6702 validation subroutine first.
- Make sure that the Waterloo Language tool does not access the 6702 when run under the 6809 emulator. It should be possible to load pre-existing application software from disk-resident files and to run them but not to edit the source code. This appears to be the effect of hacking the 6702 validation subroutine (i.e. the checksum subroutine fails with the result is that the in-built editor is designed to annoyingly misbehave). Note that microBASIC will just fail as it does not contain the same editor as the other language tools.
- Make the patch to the selected checksum subroutine second.
- Make sure that the Waterloo Language tool still does not access the 6702 when run under the 6809 emulator – but that the language tool and in-built editor now is fully functional.
- Repeat for all of the Waterloo Language Tools.

The identified patches required are presented on the last few pages of this document.

### **Test out all languages using disk-resident test programs.**

I have configured my SuperPET emulator to run the first Waterloo Software Disk Image (#1) from **disk8** and the second Waterloo Software Disk Image (#2) from **disk9/0**. This should only affect the testing of the Waterloo 6809 Development package (the linker in particular).

Please do not consider these test programs to be:

1. Good programming practice!
2. 100% tests of the Waterloo languages!


However they have proved useful to me as example programs that are known to work and do highlight admirably the situation when the 6702 is missing or not working as intended. I pass them on to you to help you test out all the Waterloo language components irrespective of whether you are familiar with the computer language concerned or not.

### **Testing EDIT:**

Invoke the '**e<RETURN>**' option from the initial ROM start-up menu.

The editor should start and you will be prompted to press the **<RETURN>** key.

Invoke the command "**g hd.txt<RETURN>**". Ten lines of "**Hello Dave**" should be loaded and displayed.



```

SP9000
<beginning of file>
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
<end of file>
hd.txt - Lines transferred = 10

```

Figure 9 - Sample output from microEditor after loading text file hd.txt

You should be able to move around within the displayed text and insert/delete/modify text lines as you wish.

Enter the command "**bye<RETURN>**" to quit from the editor and return to the ROM menu.

Don't forget that you need to be in "command mode" for the editor to take notice of the bye command!

### **Testing PASCAL:**

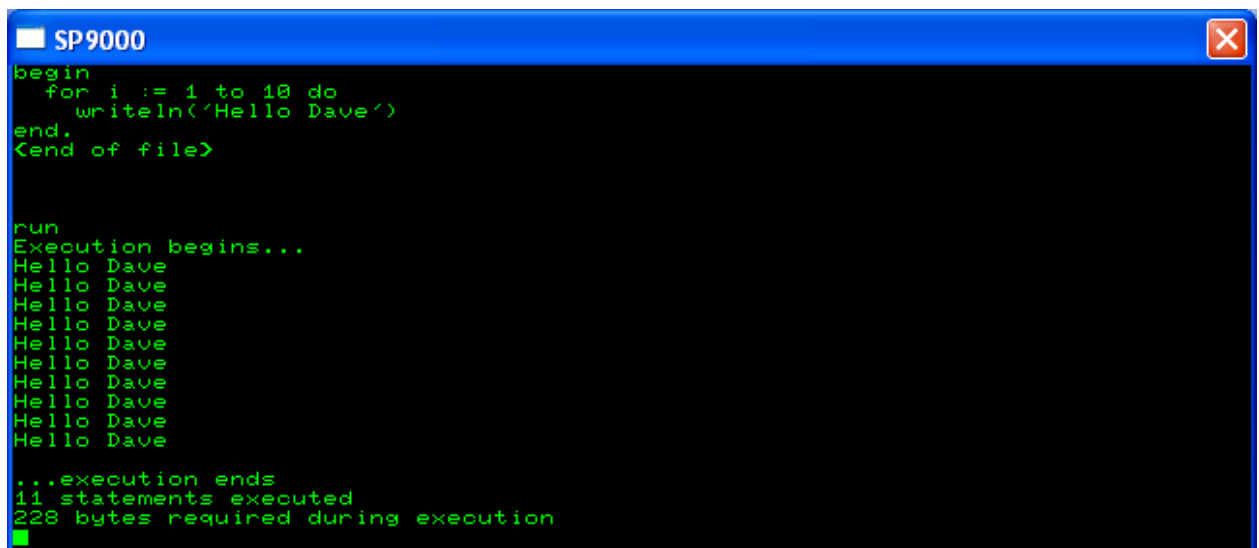
Invoke the '**p<RETURN>**' option from the initial ROM start-up menu.

The PASCAL language should start and you will be prompted to press the **<RETURN>** key.

Invoke the command "**g hd.pas<RETURN>**". This should load a test PASCAL source program from the disk into the in-built editor.

Invoke the command "**run<RETURN>**".

Ten lines of “Hello Dave” should be displayed.



```
SP9000
begin
  for i := 1 to 10 do
    writeln('Hello Dave')
  end.
<End of file>

run
Execution begins...
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
...execution ends
11 statements executed
228 bytes required during execution
```

Figure 10 - Sample output from microPascal after running program hd.pas

Press **<RETURN>** to return to the in-built editor.

Enter the command “**bye<RETURN>**” to quit from PASCAL and return to the ROM menu.

### Testing BASIC:

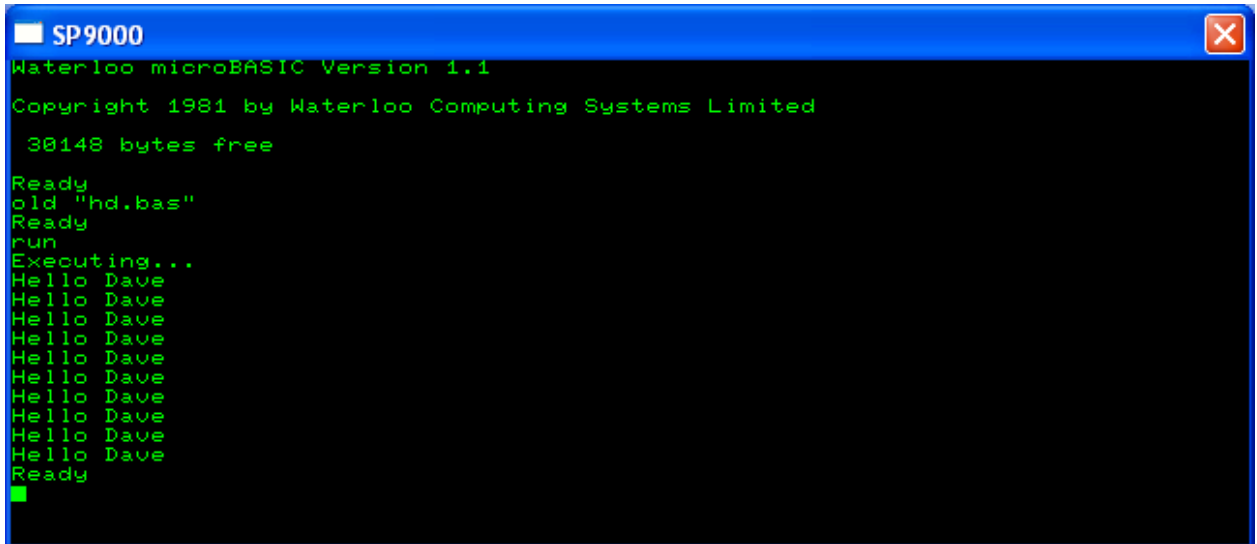
Invoke the ‘**b<RETURN>**’ option from the initial ROM start-up menu.

The BASIC language should start.

Invoke the command “**old “hd.bas”<RETURN>**”. This should load a test BASIC source program from the disk.

Invoke the command “**run<RETURN>**”.

Ten lines of “Hello Dave” should be displayed.



```

SP9000
Waterloo microBASIC Version 1.1
Copyright 1981 by Waterloo Computing Systems Limited
 38148 bytes free

Ready
old "hd.bas"
Ready
run
Executing...
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Ready

```

Figure 11 - Sample output from microBASIC after running program hd.bas

Enter the command “**bye<RETURN>**” to quit from BASIC and return to the ROM menu.

### Testing FORTRAN:

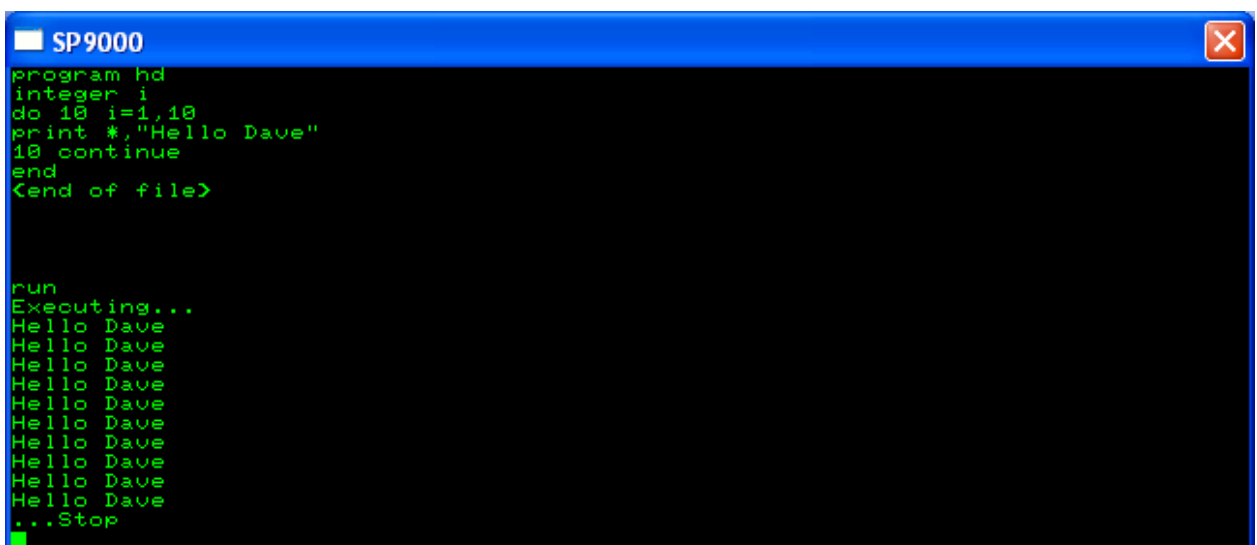
Invoke the ‘**f<RETURN>**’ option from the initial ROM start-up menu.

The FORTRAN language should start and you will be prompted to press the **<RETURN>** key.

Invoke the command “**g hd.for<RETURN>**”. This should load a test FORTRAN source program from the disk into the in-built editor.

Invoke the command “**run<RETURN>**”.

Ten lines of “**Hello Dave**” should be displayed.



```

SP9000
program hd
integer i
do 10 i=1,10
print #,"Hello Dave"
10 continue
end
<end of file>

run
Executing...
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
...Stop

```

Figure 12 - sample output from MicroFORTRAN after running program hd.for

Press **<RETURN>** to return to the in-built editor.

Enter the command “**bye<RETURN>**” to quit from FORTRAN and return to the ROM menu.

### Testing APL:

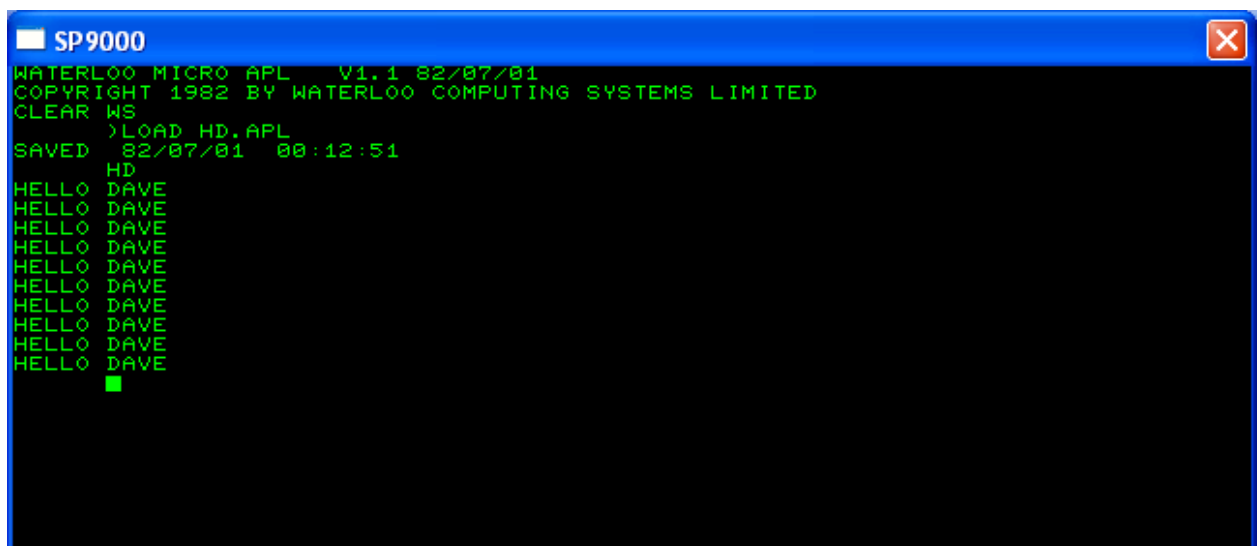
Invoke the ‘**disk9/0.a<RETURN>**’ command from the initial ROM start-up menu.

The APL language should start.

Invoke the command “**)LOAD HD.APL<RETURN>**”. This should load a test APL workspace from the disk into the in-built editor.

Invoke the function “**HD<RETURN>**”.

Ten lines of “**HELLO DAVE**” should be displayed.



```

SP9000
WATERLOO MICRO APL  V1.1 82/07/01
COPYRIGHT 1982 BY WATERLOO COMPUTING SYSTEMS LIMITED
CLEAR WS
)LOAD HD.APL
SAVED 82/07/01 00:12:51
HD
HELLO DAVE
HELLO DAVE
HELLO DAVE
HELLO DAVE
HELLO DAVE
HELLO DAVE
HELLO DAVE
HELLO DAVE
HELLO DAVE
HELLO DAVE
HELLO DAVE

```

Figure 13 - Sample output from MICRO APL after loading workspace HD.APL and executing function HD

Enter the command “**)OFF<RETURN>**” to quit from APL and return to the ROM menu.

Note that in order to use the APL language you will find it convenient to have an APL keyboard. If you do not have such a keyboard, you will find that the keys on your keyboard have been magically re-assigned! This means that the ‘)’ character is no longer above the ‘9’ on the Commodore Business keyboard – but has moved to be SHIFT ‘@’ (I think)... If you press SHIFT ‘9’ by mistake – you will get the ‘~’ character instead. APL uses a special keyboard and a special character generator ROM – meaning that there are multiple opportunities for a software emulator to get it wrong! I have specifically written my SuperPET emulator to cope with APL – but I can’t vouch for the accuracy of others.

### Testing COBOL:

Invoke the ‘**disk9/0.COBOL<RETURN>**’ command from the initial ROM start-up menu. Note that there is no short-cut displayed in the menu for COBOL and that the word COBOL must be

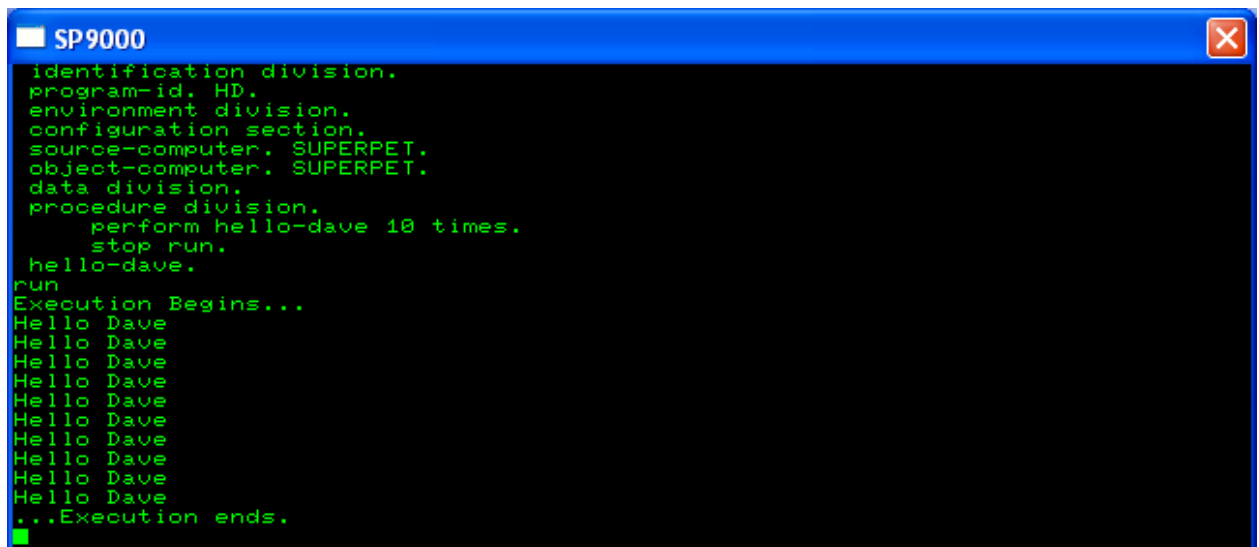
entered in full and in upper-case letters (unlike the other language processors where a single lower-case letter short-cut will suffice).

The COBOL language should start and you will be prompted to press the **<RETURN>** key.

Invoke the command “**g hd.cbl<RETURN>**”. This should load a test COBOL source program from the disk into the in-built editor.

Invoke the command “**run<RETURN>**”.

Ten lines of “**Hello Dave**” should be displayed.



```
SP9000
identification division.
program-id. HD.
environment division.
configuration section.
source-computer. SUPERPET.
object-computer. SUPERPET.
data division.
procedure division.
    perform hello-dave 10 times.
    stop run.
hello-dave.
run
Execution Begins...
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
...Execution ends.
```

Figure 14 - Sample output from microCOBOL after running program hd.cbl

Press **<RETURN>** to return to the in-built editor.

Enter the command “**bye<RETURN>**” to quit from COBOL and return to the ROM menu.

### **Testing DEVELOPMENT:**

Invoke the ‘**disk9/0.d<RETURN>**’ command from the initial ROM start-up menu. This will start the development software and will present you with a small submenu of choices:



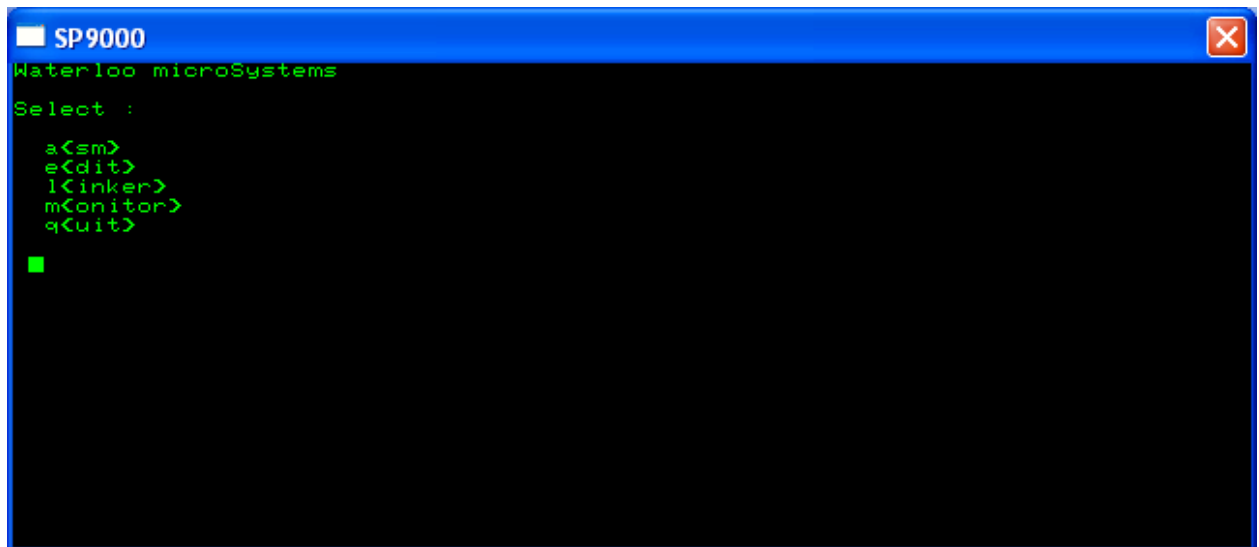


Figure 15 – Development submenu

### Invoking the development assembler:

Invoke the 6809 assembler by entering the command “**a<RETURN>**”.

The assembler should start and prompt you to enter a filename. This is the filename of a 6809 source code program. Enter the filename as “**hd<RETURN>**”.

The assembler should start to assemble the source code program (located in text file hd.asm) and produce an object file (in hd.b09) and a listing file (in hd.lst).

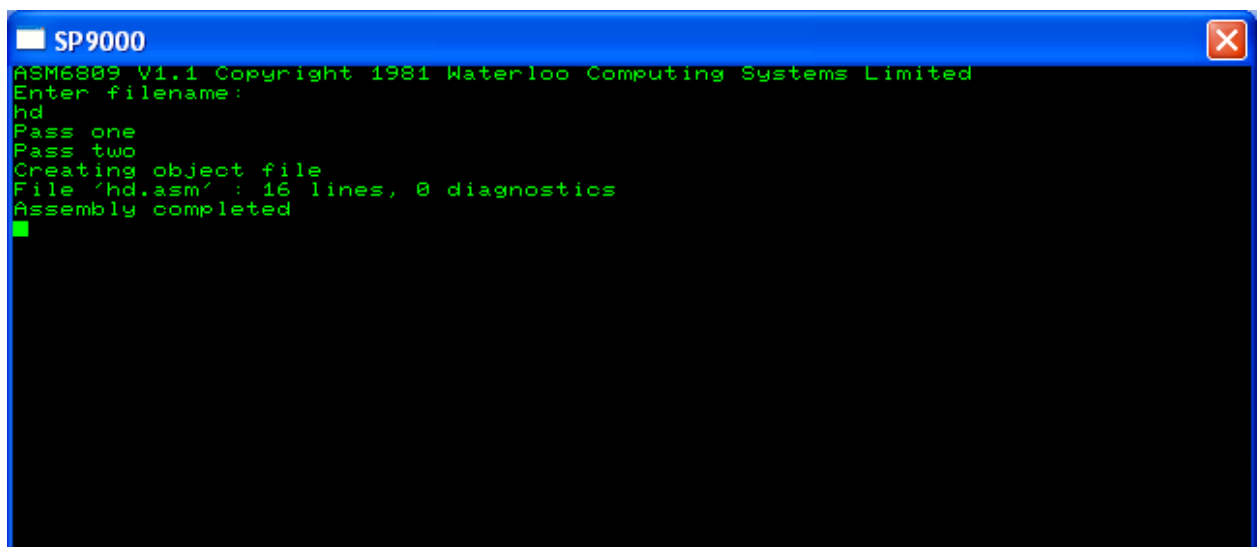


Figure 16 - Sample output from the 6809 assembler after processing source file hd.asm

When the assembly process is complete, press **<RETURN>** to exit the assembler and return to the development submenu.

### Invoking the development editor:

Invoke the development text editor by entering the command “**e<RETURN>**”.

The editor should start and you will be prompted to press the **<RETURN>** key.

Invoke the command “**g hd.lst<RETURN>**”. The assembler listing file (generated in the step above) should be loaded and displayed.



```

beginning of file>
1
2
3 0000 86 0A
4
5 0002 34 02
6 0004 CC 00 10
7 0007 BD 00 00
8 000A 35 02
9 000C 4A
10 000D 26 F3
11 000F 3F
xref printf_
lda #10
loop
pshs a
ldd #hd
jsr printf_
puls a
deca
until eq
swi
hd.lst - Lines transferred = 17

```

Figure 17 - Sample output from microEditor after loading listing file hd.lst

You should be able to move around within the displayed text as you wish.

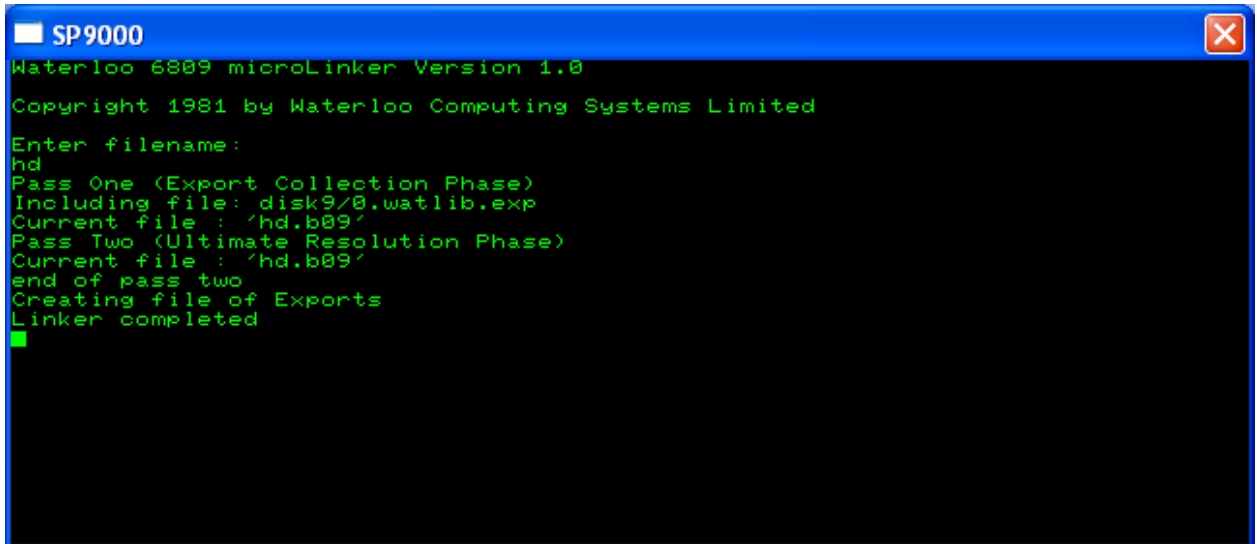
Enter the command “**bye<RETURN>**” to exit the editor and return to the development submenu.

### **Invoking the development linker:**

Invoke the 6809 linker by entering the command “**l<RETURN>**”.

The linker should start and prompt you to enter a filename. This is the filename of a linker control file. Enter the filename as “**hd<RETURN>**”.

The linker should start to link the object code program (located in object file hd.b09) with the entry points to the ROM subroutines (located in library file disk9/0.watlib.exp) to produce an executable file (in hd.mod) and a map file (in hd.map).



```
SP9000
Waterloo 6809 microLinker Version 1.0
Copyright 1981 by Waterloo Computing Systems Limited
Enter filename:
hd
Pass One (Export Collection Phase)
Including file: disk9/0.watlib.exp
Current file : 'hd.b09'
Pass Two (Ultimate Resolution Phase)
Current file : 'hd.b09'
end of pass two
Creating file of Exports
Linker completed
```

Figure 18 - Sample output from microLinker after processing command file hd.cmd

When the linking process is complete, press **<RETURN>** to exit the linker and return to the development submenu.

### **Invoking the development monitor:**

Invoke the debug monitor by entering the command **"m<RETURN>"**.

The prompt when in the monitor is the **'>** character. This character means that the debug monitor is waiting for a new command.

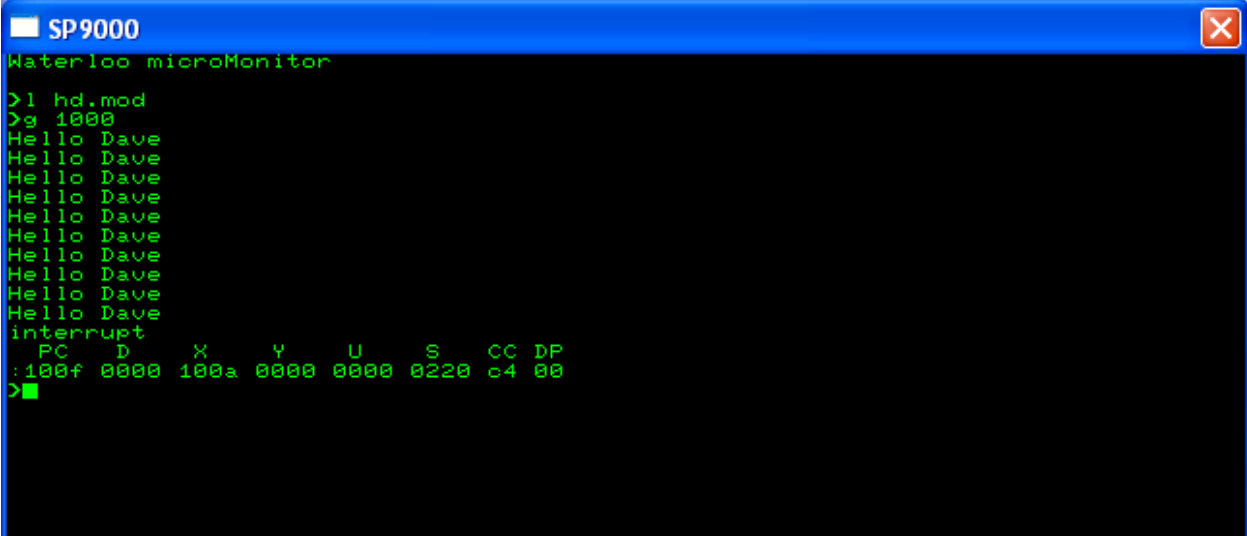
Load the executable module created by the linker by entering the command:

**"l hd.mod<RETURN>"** in response to the monitor's **'>** prompt.

Execute the executable module just loaded at the pre-defined start address by entering the command:

**"g 1000<RETURN>"** in response to the monitor's **'>** prompt.

Ten lines of **"Hello Dave"** should be displayed; followed by the word **"interrupt"**; followed by a dump of the 6809 CPU registers. The monitor should respond with the usual **'>** prompt.



```
SP9000
Waterloo microMonitor
>1 hd.mod
>g 1000
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
Hello Dave
interrupt
PC   D      X      Y      U      S      CC  DP
:100f 0000 100a 0000 0000 0220 c4 00
>■
```

Figure 19 - Sample output from microMonitor after executing hd.mod

Enter the command “**q<RETURN>**” to quit from the monitor and return to the development submenu.

Enter the command “**q<RETURN>**” to quit from the development submenu and return to the ROM menu.

This completes the test of the Waterloo Language Software.

Patches to the **second release** of the Waterloo Language disks (D64 format) to remove the 6702 dongle protection.

### **Waterloo Language Disk #1**

Language	Version	Date
BASIC	V1.1	1981
EDIT	V1.1	1982
FORTRAN	V1.1	1981
PASCAL	V1.1	1982

### **Waterloo Language Disk #2**

Language	Version	Date
APL	V1.1	1982
COBOL	V1.0	1981
DEVELOPMENT (ASSEMBLER)	V1.1	1981
DEVELOPMENT (LINKER)	V1.0	1981
DEVELOPMENT (EDITOR)	V1.1	1982

**6702 Validation Subroutine Byte Sequences of Interest**

1F 41 32 72

Needs to be patched with:

4F 5F 39 72

where: 4F is a CLRA instruction.  
5F is a CLRB instruction.  
39 is a RTS instruction.

**Checksum Subroutine Byte Sequences**

C3 FF 03 E7

Needs to be patched with:

C3 FF 52 E7

Where 52 (hex) is the data byte to make the computed checksum correct for the patches to the 6702 validation subroutine as detailed above.

The addresses of the checksum subroutines within memory have not been determined (except for EDIT) and are (as a result) presented as '????'(see overleaf). These were identified on the D64 disk image directly and I couldn't be bothered to identify them in the memory image loaded from disk.

**Waterloo Language Disk #1****6702 validation byte sequences found at:**

Disk offset in bytes (hex) from start of disk image.	Track number (decimal).	Sector number (decimal).	Byte offset within sector (hex).	Language.	Bank number when loaded into memory.	Address (hex) when loaded into memory.
86D7	T07	S08	D7	PASCAL	5	9000
113AE	T14	S02	AE	BASIC	5	93F0
18D63	T20	S02	63	EDIT	1	9852
1A820	T21	S10	20	FORTTRAN	0	960C

**Checksum byte sequences found at:**

Disk offset in bytes (hex) from start of disk image.	Track number (decimal).	Sector number (decimal).	Byte offset within sector (hex).	Language.	Bank number when loaded into memory.	Address (hex) when loaded into memory.	Address (hex) of corresponding 6702 validation subroutine.
8E89	T07	S16	89	PASCAL	5	????	9000
FD1B	T13	S01	1B	BASIC	5	????	93F0
198C9	T20	S13	C9	EDIT	1	9BB2	9852
1A7AA	T21	S09	AA	FORTTRAN	0	????	960C

Note that there appears to be a 1:1 correspondence between the number of 6702 validation subroutines and the number of checksum subroutines.



**Waterloo Language Disk #2****6702 validation byte sequences found at:**

Disk offset in bytes (hex) from start of disk image.	Track number (decimal).	Sector number (decimal).	Byte offset within sector (hex).	Language.	Bank number when loaded into memory.	Address (hex) when loaded into memory.
72C	T01	S07	2C	DEVELOPMENT (ASSEMBLER)	5	9A05
5024	T04	S17	24	DEVELOPMENT (EDITOR)	2	9852
E884	T12	S01	84	APL	6	9F12
1794C	T19	S01	4C	COBOL	0	9240
2845E	T33	S12	5E	DEVELOPMENT (LINKER)	8	94B6

**Checksum byte sequences found at:**

Disk offset in bytes (hex) from start of disk image).	Track number (decimal).	Sector number (decimal).	Byte offset within sector (hex).	Language.	Bank number when loaded into memory.	Address (hex) when loaded into memory.	Address (hex) of corresponding 6702 validation subroutine.
318A	T03	S07	8A	DEVELOPMENT (EDITOR)	2	????	9852
E879	T12	S01	79	APL	6	????	9F12
17941	T19	S01	41	COBOL	0	????	9240

Note that there are more 6702 validation subroutines than there are checksum subroutines. It would appear that the assembler and linker only validate the 6702 (i.e. there is no checksum subroutine). The checksum subroutine for the development package appears to be on the editor 6702 validation subroutine only.