David Husband, M.Sc in IT, *Baremetal Engineer Extraordinaire*

What you see detailed on my CV are "merely" the things I have done to earn money...
In reality, my abilities, knowledge and capabilities range far beyond my CV..

I am a ***BAREMETAL ENGINEER*** and what is in this document showcases a small part of my ***baremetal engineering***...
I have an embryonic website on the subject here: http://baremetal.engineer/

The purpose of that web site[1] is to expose and document some of what I have spent many years doing so that, hopefully, young engineers can & will learn from it !!

In http://baremetal.engineer/baremetal.blockchain.engineer.pdf I showed my recently acquired blockchain knowledge, including my analysis of how Blockchain can be applied to embedded IoT systems.

In http://baremetal.engineer/baremetal.hardware.engineer.pdf I showed how electronic hardware has not changed in essence over the last 60 years or so, due to being based upon the principles of physics established hundreds of years ago. I showed how, counter-intuitively, electronics hardware was actually ***easier now*** due to ever increasing integration and increasing functionality...

I also have extensive radio-communications knowledge and ability[2] (having been a licensed radio amateur since 1973); having successfully designed, manufactured, and sold worldwide, a range of products to decode various kinds of data transmitted over radio... And I know what an SDR is! https://en.wikipedia.org/wiki/Software-defined_radio

See these sample images: Image# 56, Image# 57 & Image# 58

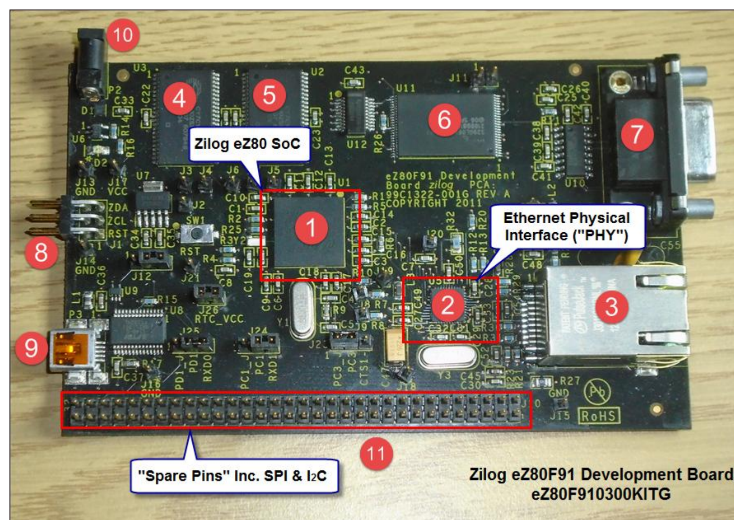
To be brief here, I have created a number of Appendices where I go into further detail on particular topics[3]

In this document, I present an extract of what I have been doing over the last six months or so, which is designing, writing and testing some ***low-level hardware interrupt drivers and tasks in assembler ("machine code")*** to enable the use of the Ethernet interface on Zilog's eZ80 System-on-a-Chip using Zilog's eZ80F91 development platform... See Image# 1 below. The purpose of this document is to "showcase" a sub-set of my skills and abilities[4]...

This is not trivial because the eZ80's ***Ethernet Media Access Controller*** ("***EMAC***") is a very complex peripheral which has to communicate with the Ethernet outside world via a complex ***Ethernet Physical Interface*** ("PHY")

Given the complexity of both these interfaces/devices, it takes a great deal of skill and persistence to get them working correctly from "scratch" on any system... (See ***Complexity or Simplicity?***)

See: http://baremetal.engineer/eZ80F91_EMAC.pdf & http://baremetal.engineer/New_PHY_ICS1894-40.pdf



Image# 1 - The Zilog eZ80F91 (eZ80F910300KITG) Development Board

Interrupts and the associated Tasks I get them to generate, are implicitly ***An Event-Driven Environment***, & rather hardware-centric

The EMAC part of the eZ80 has 8k of fast DMA RAM, 4k for receiving packets and 4k for transmitting packets...

---

[1] You will see from that URL that my skills & abilities do not extend to scripting web pages !!

[2] I am currently setting up my own satellite transmitting/receiving terminal to use the new Geostationary QO-100 Amateur Radio Transponder ***https://amsat-uk.org/satellites/geo/eshail-2/*** Web-SDR for QO-100 at Goonhilly: ***https://eshail.batc.org.uk/nb/*** ***https://eshail.batc.org.uk/wb/***

[3] I've recycled some images, explanations, etc, from my Master's Dissertation from around 2011-2012...

[4] And my recent work described here with the eZ80's EMAC, starting by implementing & testing low-level TCP/IP layers such as the ***Address Resolution Protocol*** ("***ARP***") and the associated ***ARP Cache***, are ***totally new knowledge for me & indicate my on-going ability to acquire & apply new knowledge & skills***...

The EMAC internally handles the "dirty work" of the basic Ethernet frames and their checksums and creates a "descriptor table" for each received packet so a number of house-keeping tasks are performed. Initially an EMAC register is loaded with the MAC address ("Hardware Address") of the host and enabled to generate an interrupt upon the reception of a Broadcast Packet and/or a packet addressed to the hardware address the EMAC has been setup with...

The EMAC also needs to be told what packet "block size" to work to and I have chosen to use blocks of 256 bytes... So if an ARP Packet is received, its size is always 64 bytes which the EMAC reads into a 256 byte block in its DMA RAM and then generates an interrupt...

Other packets such as TCP/IP packets with a type of 800 hex can vary in size, up to 3 or more 256 byte blocks before an interrupt is generated...

The EMAC Rx 4k DMA packet buffer is a recirculating buffer which needs to be processed as quickly as possible. What I do is to read each received packet into my own recirculating Packet Buffer of 64k of Ram and then generate a ***buffer reading task*** and then exit the ***Rx Packet interrupt service routine***...

The interrupt service routine needs to be as fast and as short as possible and to do the very minimum required. Generally, just to read the appropriate data into a circular RAM buffer, and maintain a "***putting-in***" ***buffer pointer***, manage any registers associated with the interrupt and set an ***"interrupt buffer processing" task*** (which will, in due course be run in the background, outside of the interrupt path) ...

Tasking in a closed system, i.e. where the task binding is early and is at compile time can be very simple and robust... I use just **ONE BYTE** called "***TASKS***" to hold pending tasks and a routine called "***PAUSE***" to manage and execute tasks. The size of ***PAUSE*** is only 54 bytes so it is compact and FAST... It is not able to be optimised !!

The use of ***PAUSE*** within The ***Forth Virtual Machine*** ("VM") and other looping structures is described in more detail here: ***Multi-tasking***

So, although the work described is in ***eZ80 assembler***, it is done within a modified FIG-Forth model/environment. See ***The Forth Paradigm***

Forth is an ***extensible Interactive Compiler*** where ***the act of programming to solve a problem is to extend the word-set of the language***... Invented by Charles Moore

> "The solution in Forth is not arrived at by writing programs, but by creating a new instruction set in the Forth *virtual computer*. The new instruction set in essence becomes *the solution* to the programming problem. This new instruction set can be optimised at various levels for memory space and for execution speed, including hardware optimisation. Forth allows us to surpass the fundamental limitation of a computer, which is the limited and fixed instruction set. This limitation is also shared by conventional programming languages, though at a higher and more abstract level"

Image# 2 - The Forth approach to programming. Image: Husband, 2011, based on Ting (1989, p. 10)

So to put it another way, a problem is solved in Forth by extending the Forth word-set in contrast to C or all other high-level languages which ***force you to fit the problem into the word-set and to the syntax***[5] ***of the language*** being used...

Now let's turn to the subject of how I test and develop my code... Over the years, I have devised a system that leverages some of the unique aspects of developing/testing code on a Forth System...

This my variation of ***TEST-DRIVEN DEVELOPMENT*** and based upon ***The Forth Paradigm***...

When I am ***developing in assembler***, I do this in at least ***three ways*** which are all perfect examples of what the right-hand side of Image# 47 is showing !!

Method#1

For my first example, I will describe how I started testing the ***ARP Cache***...

Ethernet nodes on the network transmit "***ARP Requests***" as ***Broadcast messages*** because the ***TCP/IP protocol encapsulated within Ethernet packets*** uses its own ***32-bit IP addresses***, whereas an Ethernet network uses ***48-bit***

---

[5] ***Syntax creates nasty unintended consequences***; the more elaborate the syntax, the more error checking that can be done, but the more human errors that will be flagged – ***the programmer then becomes a slave to the compiler***; the problem is the arcane, arbitrary, and cryptic syntax of most languages, which ***must accommodate all of the intended [future] applications***; that makes the compiler much more elaborate.... My emphasis; based on (Biancuzzi & Warden, 2009, p. 65) citing Charles Moore, the inventor of Forth

"***Hardware Addresses***" a.k.a. "***MAC Addresses***" and sending an ***ARP Request*** is the only way to establish the mapping between the two addressing systems... An ***ARP Cache*** is employed to greatly reduce the number of ***ARP Requests*** so as to reduce network load...

I decided that as my setup was for a very small network, to set the ***ARP Cache*** size at 32 elements, with each element having 11 bytes. See Image# 3 below...

My next step was to create a Forth word called "***ARP_CACHE***" which is invoked from the Forth System by merely typing its name and pressing "Enter"... See Image# 4 below...

When the system is running with the Ethernet cable connected to my Netgear Ethernet Switch, packets are automatically received and decoded in the background by the EMAC Rx Packet Interrupt handler, ***EmacRxIrq***, which triggers a task, ***TASK0***, which executes after ***EmacRxIrq*** has finished and calls ***DO_CACHE*** to manage the ***ARP Cache***...

ARP_CACHE is used after a test run of the system with the Ethernet cable being disconnected (See Image# 15 Point 1) after sufficient packets have been captured. There is more on this in Image# 4 above, the latest version of Forth test-word ***ARP_CACHE*** has been run and is presenting the expected results... In Image# 5 above, a capture from an earlier version of ***ARP_CACHE*** is shown, displaying a number of bugs in ***DO_CACHE***..

(1) Duplicates are being saved... This is to be expected as the ***ARP_CACHE*** test-word was written ***before*** the full functionality was implemented in ***DO_CACHE***... ***Test Driven Development !!***

(2) One of the items is showing an IP Address of 0.0.0.0 -- I thought this might be a bug in my s/w, but I Googled and found it was known as a "***Gratuitous ARP Request***" but opinions varied and none accounted for the 0.0.0.0 IP address, so I decided to filter it out entirely...

The ***ARP_CACHE*** word is relatively simple for me, so I did not expect it to have any bugs... However, I did see a "***feature***" so I swapped the order in which the Cache data was presented (Image# 5) so that there would be no "ragged edges" displayed as seen in Image# 5 below

This tests if ***DO_CACHE*** is extracting the ***ARP Request's*** "***Source IP Address***" and "***Source Hardware Address***" and then storing them correctly in the ARP Cache's 32x11 byte memory array...

```
040000 00000000  ArpCache1:  DB    0,0,0,0      ; Source IP Address
040004 00000000 0000         DB    0,0,0,0,0,0  ; Source h/w Address
04000A 00                    DB    0            ; Item Flags
04000B           ArpCache2:  DS    11
040016           ArpCache3:  DS    11
040021           ArpCache4:  DS    11
04002C           ArpCache5:  DS    11
040037           ArpCache6:  DS    11
040042           ArpCache7:  DS    11
040129           ArpCache28: DS    11
040134           ArpCache29: DS    11
04013F           ArpCache30: DS    11
04014A           ArpCache31: DS    11
040155           ArpCache32: DS    11
    00040160     ArpCacheEnd: EQU  $
040160 0100      ArpScore    DW    1
040162           ArpCachePtr DS    3
040165           ArpCacheCtl DS    1
```

Image# 3 - The source code for defining the ARP Cache array I decided to implement

So in Image# 4 below, ***ARP_CACHE*** is showing the correct operation of the ***ARP Cache***. It should be noticed that the Ethernet Node 90:B1:1C:78:68:82 is miss-configured to IP 169.254.43.10 !!

At an earlier stage in the testing & development of ***DO_CACHE***, that miss-configuration was causing ***DO_CACHE*** to enter an endless loop... It suited my purposes not to fix the miss-configuration at that point because it was a good test of my subsequent bug-fix !!

Summary:

Method#1 is also a form of static-testing, where perhaps you run the test-word before the software-under-test ("***S.U.T***"), and/or you run it afterwards and analyse the results... You then fix any bugs and re-run the test !!

So here, the new Forth Testing word called ARP_CACHE is executed from the keyboard by typing its name followed by the [Enter] key...

... and this is ARP_CACHE's output to the terminal screen...

```
ok
ok
ARP_CACHE
040000  01  00:90:A9:77:A7:CB  192.168.1.130
04000B  01  90:B1:1C:78:68:82  169.254.43.10
040016  01  90:B1:1C:5E:D6:DF  192.168.1.40
040021  01  B8:CA:3A:9E:D0:2B  192.168.1.4
04002C  01  9C:AE:D3:ED:F6:D0  192.168.1.128
040037  00  00:00:00:00:00:00  0.0.0.0
040042  00  00:00:00:00:00:00  0.0.0.0
04004D  00  00:00:00:00:00:00  0.0.0.0
040058  00  00:00:00:00:00:00  0.0.0.0
040063  00  00:00:00:00:00:00  0.0.0.0
04006E  00  00:00:00:00:00:00  0.0.0.0
040079  00  00:00:00:00:00:00  0.0.0.0
040084  00  00:00:00:00:00:00  0.0.0.0
04008F  00  00:00:00:00:00:00  0.0.0.0
04009A  00  00:00:00:00:00:00  0.0.0.0
0400A5  00  00:00:00:00:00:00  0.0.0.0
0400B0  00  00:00:00:00:00:00  0.0.0.0
0400BB  00  00:00:00:00:00:00  0.0.0.0
0400C6  00  00:00:00:00:00:00  0.0.0.0
0400D1  00  00:00:00:00:00:00  0.0.0.0
0400DC  00  00:00:00:00:00:00  0.0.0.0
0400E7  00  00:00:00:00:00:00  0.0.0.0
0400F2  00  00:00:00:00:00:00  0.0.0.0
0400FD  00  00:00:00:00:00:00  0.0.0.0
040108  00  00:00:00:00:00:00  0.0.0.0
040113  00  00:00:00:00:00:00  0.0.0.0
04011E  00  00:00:00:00:00:00  0.0.0.0
040129  00  00:00:00:00:00:00  0.0.0.0
040134  00  00:00:00:00:00:00  0.0.0.0
04013F  00  00:00:00:00:00:00  0.0.0.0
04014A  00  00:00:00:00:00:00  0.0.0.0
040155  00  00:00:00:00:00:00  0.0.0.0
ok
```

Image# 4 - The new Forth test word ARP_CACHE in action !

These are all duplicates, and there must be no duplicates !

This is a "Gratuitous" ARP Request

IP Address

Hardware Address or MAC Address

```
ARP_CACHE
040000 192.168.1.131  A0:CE:C8:05:21:A2  00
    0B 192.168.1.133  90:B1:1C:78:68:82  00
    16 192.168.1.130  00:90:A9:77:A7:CB  00
    21 192.168.1.133  90:B1:1C:78:68:82  00
    2C 192.168.1.133  90:B1:1C:78:68:82  00
040037 192.168.1.130  00:90:A9:77:A7:CB  00
040042 192.168.1.133  90:B1:1C:78:68:82  00
04004D 192.168.1.130  00:90:A9:77:A7:CB  00
040058 192.168.1.133  90:B1:1C:78:68:82  00
040063 192.168.1.133  90:B1:1C:78:68:82  00
04006E 192.168.1.3  CC:40:D0:11:18:C3  00
040079 0.0.0.0  A0:CE:C8:05:21:A2  00
040084 192.168.1.131  A0:CE:C8:05:21:A2  00
04008F 192.168.1.131  A0                 00
04009A 192.168.1.131  A0                 00
0400A5 192.168.1.131  A0:CE:C8:05:21:A2  00
0400B0 192.168.1.133  90:B1:1C:78:68:82  00
0400BB 192.168.1.131  A0:CE:C8:05:21:A2  00
0400C6 0.0.0.0  00:00:00:00:00:00  00
0400D1 0.0.0.0  00:00:00:00:00:00  00
0400DC 0        00:00:             00
0400E7 0.0.0.0  00:00:             00
0400F2 0.0.0.0  00:00:00:00:00:00  00
0400FD 0.0.0.0  00:00:00:00:00:00  00
040108 0.0.0.0  00:00:00:00:00:00  00
040113 0.0.0.0  00:00:00:00:00:00  00
04011E 0.0.0.0  00:00:00:00:00:00  00
040129 0.0.0.0  00:00:00:00:00:00  00
040134 0.0.0.0  00:00:00:00:00:00  00
04013F 0.0.0.0  00:00:00:00:00:00  00
04014A 0.0.0.0  00:00:00:00:00:00  00
040155 0.0.0.0  00:00:00:00:00:00  00
ok
```

Image# 5 - An earlier version of ARP_CACHE running showing a number of bugs !!

Method#2

This where you embed calls to a number of ***test-displays*** which display appropriate internal information, state, etc, while the "***software-under-test***" is running... I show an example of this below...

This is done by making a call to a debugging display routine, in this case, ***ETH_DEBUG***, as shown in Image# 7 below... This is done from CACHE_LOOP as shown in Image# 6 below with the A Register just before the call containing a value that determines which ETH_DEBUG routine is invoked as appropriate to suit the calling program. See Points 2,3,4 below

Whether or not ***ETH_DEBUG*** does anything is controlled by the state of the bits in ***CONTRL***, (Point 5) so early in the development/debugging cycle, ***ETH_DEBUG*** can be enabled and then later can be disabled[6]...

At present, the bit position in the A Register determines which ***Ethernet Debugging Display*** is executed, and there is no way to select which ones are or are not, in the same way that the bits of ***CONTRL*** can be manipulated. This will be updated in later iterations

```
010A83         (1) CACHE_LOOP:
              ;------------------------------ Test Cache Item -------------------
010A83 3E01        (2) LD    A,00000001B ; Test Cache Item Display
010A85 CD BA 0B 01     CALL  ETH_DEBUG
              ;---------------------------- See if IP addresses match ------------
010A89 3E10        (3) LD    A,00010000B ; "Is it a Duplicate?" #1 Display
010A8B CD BA 0B 01     CALL  ETH_DEBUG
010A8F FD2723          LD    HL,(IY+35)  ; HL points to Source IP addr in Pkt
010A92 DD1700          LD    DE,(IX+0)   ; DE contains the contents of ARP Cache
                                         ;  pointed to by IX
010A95 B7              OR    A           ; Clear Carry Flag
010A96 ED52            SBC   HL,DE       ; Do they match?
010A98 C2 D5 0A 01     JP    NZ,DO_CAC2  ; No
010A9C 3E20        (4) LD    A,00100000B ; "Is it a Duplicate?" #2 Display
010A9E CD BA 0B 01     CALL  ETH_DEBUG
```

Image# 6 - How CACHE_LOOP invokes ETH_DEBUG...

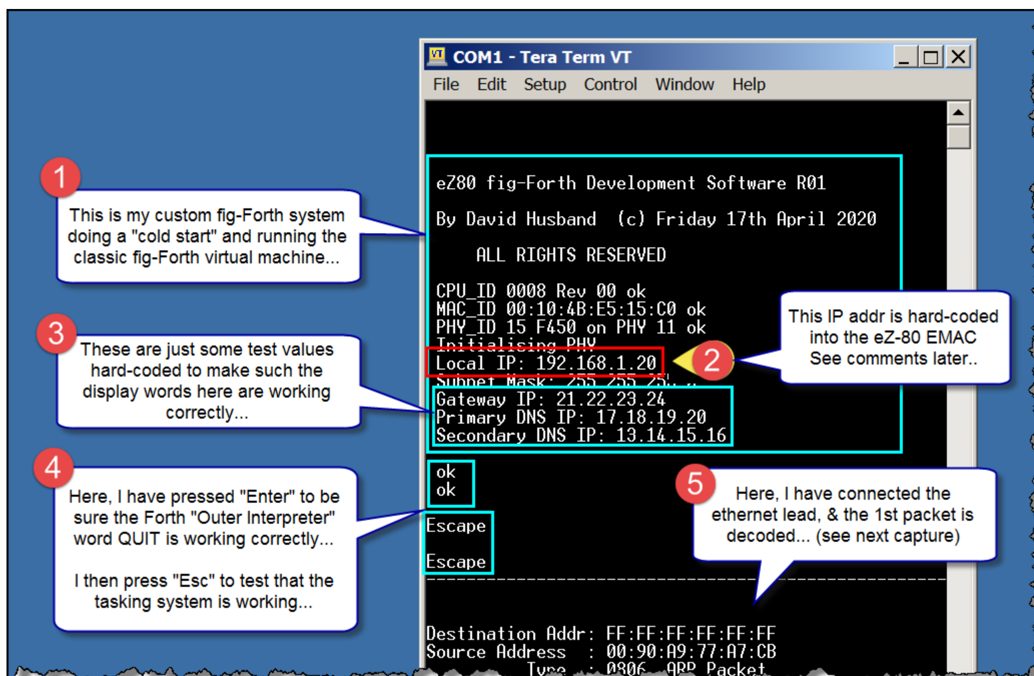
```
              ;---------------------------- Ethernet Debug Display --------
010BBA              ETH_DEBUG:
010BBA E5               PUSH   HL
010BBB 4021F6FF     (5) LD.SIS HL,CONTRL
010BBF CB5E             BIT    3,(HL)
010BC1 CA C7 0B 01      JP     Z,ETH_DE0
010BC5 E1               POP    HL
010BC6 C9               RET
010BC7              ETH_DE0:
010BC7 CB47             BIT    0,A
010BC9              (6) JR     Z,ETH_DE1
010BCB        01        LD     HL,LARP0     ; "Test Cache Item"
010BCF        00        CALL   MSG
010BD3 C3 3A 0C 01      JP     ETH_DE_END
010BD7              ETH_DE1:
010BD7 CB4F         (7) BIT    1,A
010BD9 28 0C            JR     Z,ETH_DE2
010BDB 21 C1 0C 01      LD     HL,LARP1     "Populate"
010BDF CD 9A 42 00      CALL   MSG
010BE3 C3 3A 0C 01      JP     ETH_DE_END (8)
```


Image# 7 - How ETH_DEBUG decides which display to use... (It tests the A Register)

So the code in ***ETH_DEBUG*** in Image# 7 above (Points 6 & 7) "ripples through" testing the contents of the A Register. Jumps to ***ETH_DE_END*** shown at Point 8 above invoke a "common stub" that currently just displays some information common to all the debugging display calls...
This can be seen in operation in Point 11 of Image# 9 below...

The advantage of doing it this way means the ***debug display*** calls in ***CACHE_LOOP*** above never need to change!

[6] Given that the intention is to place all of this into the public domain so that young engineers can learn from it, it would be useful to be able to turn-on or off the debugging displays as they show useful learning/operational information, too... (So the intention is to leave it in permanently, but allow it to be controlled from the k/b)

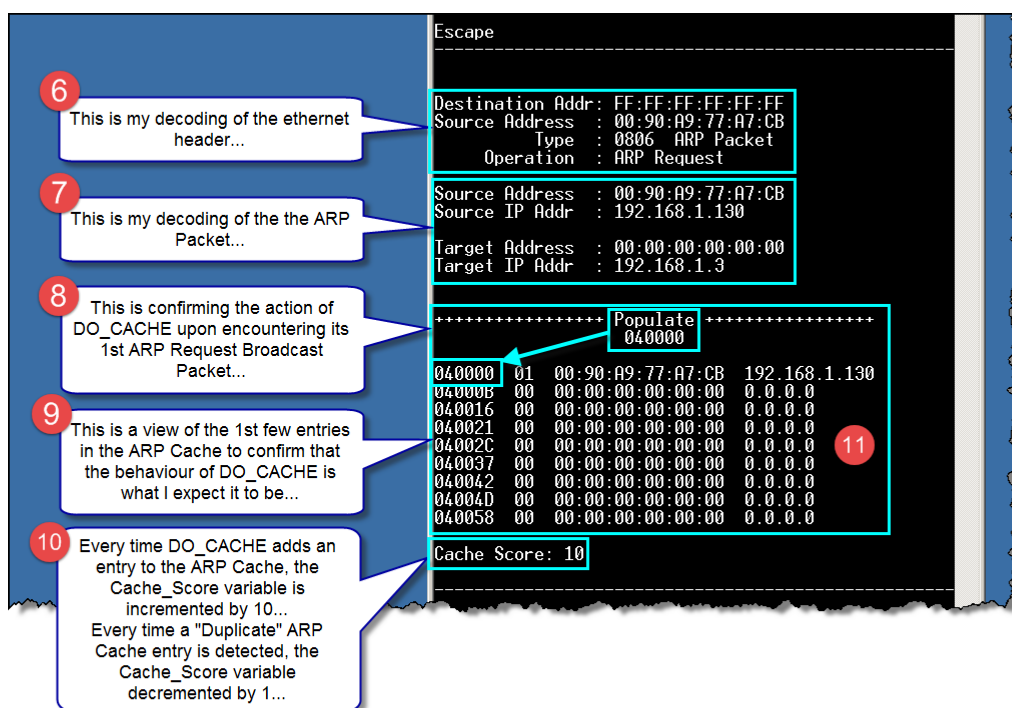


Image# 8 - Initial Power-up of the system

The ***MAC_ID*** ("***Hardware Address***") shown just above in Image# 8 - Step 2, is written into the six-byte ***EMAC Station Address Register*** to identify the node and the software also associates the IP address in Step 2 with the node

Between Steps 4 & 5, I have connected the Ethernet cable into my network switch (Image# 15) and packets are being received by the EMAC and the Ethernet Rx Interrupt is now processing packets correctly...

This can be seen in Image# 9, step 6, where the Ethernet header is decoded and it is known to be an ***ARP Request Packet***, and where its contents are decoded in Step 7



Image# 9 - Decoding the 1st ARP Request Packet

The information we are wanting to store in the ***ARP Cache*** is the ***Source Address*** and the ***Source IP Address*** as this is the ***mapping that needs to be buffered in the cache*** to avoid extra Ethernet traffic. This is known at Step 8 where ***DO_CACHE*** is called. DO_CACHE enters a 32-iteration loop starting at CACHE_LOOP (the start of which is seen in Image# 6)

At Image# 9, step 11 above, is the debugging display produced by the code at label ***ETH_DE1*** of Image# 7 above...

So the techniques described here in ***Method#2*** are ideal for debugging real-time event-driven assembler code within a Forth interpretive environment

Method#3

This can be used from assembler or high-level Forth code... I normally us it in a "***crash-and-burn***" mode, where I invoke it from my "***point-of-interest***", it captures what I want it to, and then goes into an endless loop (effectively "crashing the system") and then I can analyse the data it has captured and compare it against my expectations of the code behaviour...

The debugging tools provided as part of the ZDS-II IDE by Zilog are not really adequate for debugging a virtual machine, as they seem to be more suitable for debugging "C" Programs, and no surprise as Zilog supplies a C Compiler as part of its free development support for its eZ80 system

Given that the Forth system uses a simple ***Virtual Machine*** ("***VM***") and ***indirectly-threaded code***, I had to write my own debugger[7] to be able to support this kind of operation...

The eZ80 has two modes of operation -- a 16-bit mode which is "classic" Z80 and an extended 24-bit mode, and Zilog support this operation by having "extended" versions of "classic" Z80 registers, apart from the AF register...

It is not "either/or"; there is a mixed-mode of operation that I use, and so whether a register, say DE, appears as 16-bit register DE or as the 24-bit register DE.L will depend upon the code context, i.e. they are the same registers. See Image# 10, Items 11 & 12 below, and Items 9 & 10

The only exception to this are the two stack pointers which are separate and independent registers because there are two separate and independent stacks. You have the classic Z80 16-bit stack pointer in Item 1 and then there is an extended stack pointer for a second, 24-bit stack in Item 2

Because Forth is a stack-based system, what all the stacks contain is of interest during debugging and so these are exposed as seen in Items 3,4 & 6 See also ***The Importance of the Forth Stack***

Because the Forth I use is ***indirectly-threaded***, the contents of a memory location pointed to by the contents of a register is of interest...

```
---------------------------------------- 24-Bit Debugger ----------------------------------------
  (8)                                         (1)       (9)      (5)
  AF      BC       DE   (11)  HL       IX       IY       SP       PC      (RPP)     I     1 0 0 0 0 1 1 1
 A887    0071     01A8       0000     0000     0100     FF65     0ABB     E2F8    0100   S Z   H   P N C
(FFFF)  (03ED)   (FFFF)     (FFFF)   (0000)   (00CA)   (FFFF)   (18FE)   (00E0)
                                                      (2)                                       (7)
         BC.L     DE.L  (12) HL.L     IX.L     IY.L     SP.L     PC.L (10)
        000071   8201A8    820000   040000   0C0000   13FFE4   010ABB
       (03ED39) (FFFFFF)  (FFFFFF) (000082) (00C900) (BB0A01) (18FEED)

L-STACK BASE> 02C102 003871 000B91 00FFF7 005555 005555 010A3F 0C0200 (3)

P-STACK BASE> E000 0B95 8142 3871 0B91 FF81 5555 5555 (4)

R-STACK BASE> 023E 38F5 E050 E000 (6)
```

This is the contents of the Flags Register ("F Reg") The "F" reg part of the "AF" register...

Image# 10 - The 24-bit Debugger, ADL_REGS

The Importance of the Forth Stack

The ***Forth Stack*** is fundamental to the whole operation of Forth and is available interpretively to the user keyboard. It is used to pass values ("***parameters***") to and from Forth words and is therefore known as the "***Parameter Stack***" and is shown above in Image# 10 as the "***P-STACK***". It is 16-bits in size and is actually the microprocessor's stack[8]

The ***Parameter Stack*** operates implicitly from the keyboard because of the simple parsing behaviour of Forth's "***Outer Interpreter***"[9] ***QUIT***

The Simplicity (& power) of the Parser[10]

Forth works in any number base[11], and it defaults to base 10, so when you type a number as a text string and press "***Enter***", the parser looks up that text string in the Forth dictionary[12]. If it does not find it in its dictionary, it tries to

---

7 Not quite as daunting a task as it may seem!

8 Another powerful, implicit debugging feature!

9 Forth's "***Shell***", of which you could have more than one! [***https://en.wikipedia.org/wiki/Shell_(computing)***](https://en.wikipedia.org/wiki/Shell_(computing))

10 [***https://en.wikipedia.org/wiki/Parsing***](https://en.wikipedia.org/wiki/Parsing)

11 Providing you can represent the symbols in ASCII!

12 A singly linked-list containing about 95% of Forth

convert it from a number text string into an integer number[13] according to the number base. If it is successful, it places that number onto the ***Parameter Stack***. If unsuccessful it flags an error. (See ***Complexity or Simplicity?***)

Implicit Test Scaffolding & Unit Testing

Given what I said about the stack in the previous section, it must therefore be obvious that Forth is inherently capable of Test Scaffolding and Unit Testing. This is something I am very mindful of when I am developing & debugging code! I know that I can feed my new Forth words with various values to perform testing!

Given Forth's interactive & modular nature, test scaffolding is easy to implement, most times just involving entering a word (from the keyboard) with the correct stack contents and then observing the execution behaviour of the word and its subsequent effect upon the stack. If necessary, text can be output to the debug console and/or the register dump utility used, both to provide further information to analyse

Method#4

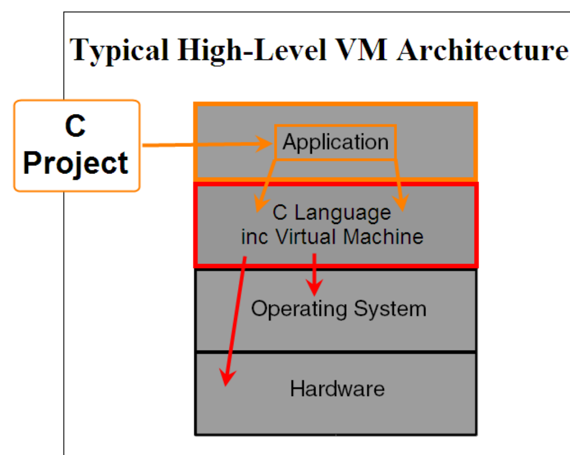
I use a high-level ***Forth decompiler*** ("***SPELL***") I have developed as a powerful static code inspection/analysis tool, mainly for when I write new compiler words so that I can see if it is creating the code I expect it to create...
See: ***Static Analysis using "SPELL"***

My Attitude to bugs

My approach is to test and debug while developing using the methods described above and not to allow bugs to fester !! See: ***Death to Bugs !!***

Back to other languages!

As shown in Image# 11 below, a typical C (or C++ or even Java) application sits on top of the language implementation which in turn sits on top of the operating system which sits on top of the hardware...



Image# 11 - A Typical C Application

Image: Husband, 2011, based on (Taivalsaari, 2003, p. 10)

Programming the C application consists of writing routines by invoking the various commands available within the syntax (rules) of the language; in the case of C by defining, declaring and using ***functions*** and ***variables...*** ***Arguments*** are used to communicate data between functions

The C language is responsible for communicating to and from the operating system and/or the host hardware. The application must direct everything via the C language it is written in (and within its fixed, complex syntax)

In C, the operating system, the hardware and any associated data are therefore purposely hidden from the application and the programmer!![14]

Other languages work the same way

The Forth application architecture is shown below in Image# 12. Programming an application in Forth is radically different to how you operate in other languages..

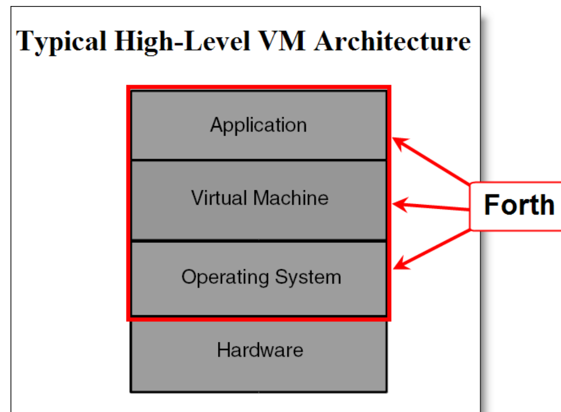
---

[13] This will be either a 16-bit fixed-point binary number or a 32bit double fixed-point binary number...

[14] And therein lies ***the real problem***... Any high-level language that hides the operating system, the hardware and any associated data from the user (and the application) ***is not ideal for the programming and/or testing of Embedded Systems***. Forth does none of these things - quite the opposite!

"Forth is a computer language with minimal syntax. It features an explicit parameter stack that permits efficient subroutine calls. This leads to ***postfix expressions*** *(operators follow their arguments*[15]*) and encourages a highly factored style of programming with many short routines sharing parameters on the stack"* (Biancuzzi & Warden, 2009, p. 60) citing Charles Moore

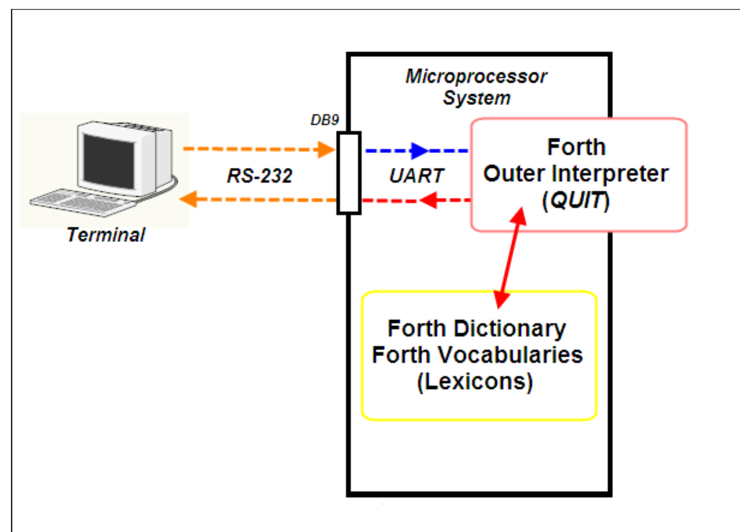
"One doesn't write programs in Forth. ***Forth is the program****. One adds words to construct a vocabulary that addresses the problem. It is obvious when the right words have been defined, for then you can interactively solve whatever aspect of the problem is relevant"* (Biancuzzi & Warden, 2009, p. 66) citing Charles Moore (My emphasis)



Image# 12 - How Forth is more than just a Virtual Machine.

Image: Husband, 2011, Based on Taivalsaari (2003, p. 10)

The "Classic" Forth Embedded Model



Image# 13 - A Typical "Classic" Embedded Forth System (Now outdated)

Image: Husband, 2011

This was the relatively very simple FIG-Forth model promoted very successfully in the early 1980's by the Forth Interest Group ("FIG"), and although it is still a valid system, it has very limited uses nowadays and would be completely useless to implement any ***Internet of Things*** devices[16]...

The "New Model" eZ80 Embedded Forth System

I am sure we are all familiar with how most embedded systems work nowadays with their user-interface operations being based upon ***talking to them via an Internet browser program*** and their support for various USB devices...

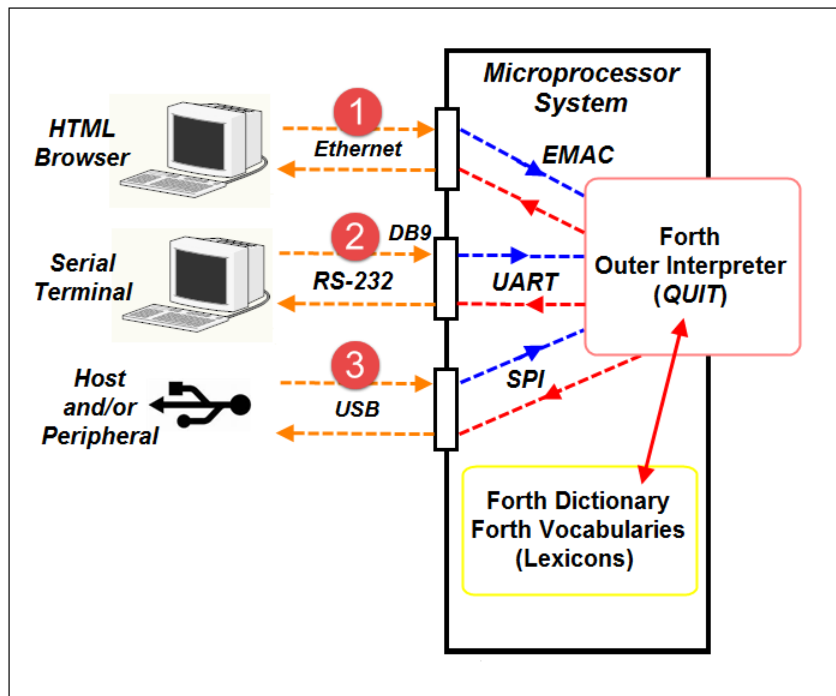
---

[15] In English we say "***Red House***"; in French they say "***Maison Rouge***" -- "***House Red***".... Reverse Polish Notation is quite a straightforward way of operating and makes quite a few things much easier... Remember: ***Complexity or Simplicity?***

[16] Because it has no USB or Network Connectivity, either Ethernet or Wi-Fi

So, "***talking to them via an Internet browser program***" all sounds so simple and straight-forward, but is technically quite complex... Your embedded system needs to run as its own website, serving or "pushing" dynamic web pages over an Ethernet interface via the HTTP & TCP/IP protocol[17],

The system outlined in Image# 14 below is the system I am working towards at the moment[18].



Image# 14 - A "New Model" eZ80 Embedded Forth System

Because the ***Evaluation Platform*** ("***EP***") contains hardware & software Ethernet support via a PHY interface and an internal EMAC, I am concentrating upon implementing as much functionality as I can using the EP

Given the difficulties in prototyping SMD-based devices, I am inclined to design my own "***development expansion platform***", connecting to a short ribbon-cable that would plug-in to the "spare pins" interface shown in Point 11 of Image# 1 & Point 10 of Image# 15 below... It would be used to implement the hardware functionality of a USB interface as shown in Point 3 of Image# 14 above... And maybe even a CAN Bus interface...

USB, by its very nature is quite complex and at an early stage, design decisions[19] must be made.
The MAX3421 USB Peripheral/Host Controller which interfaces to an SPI interface is my favourite, but for a product development platform, you'd want two USB interfaces using the MAX3421 controller, because you might well want/need to implement a Host USB ***and*** a Peripheral USB

Last but not least -- Configuration Mgt - Another VITAL Software Engineering Management Tool !!

I now use AllChange to manage my configuration management as an integral part of my software development & testing process... See: ***Configuration Management - Baselines & Versioning***

Summary

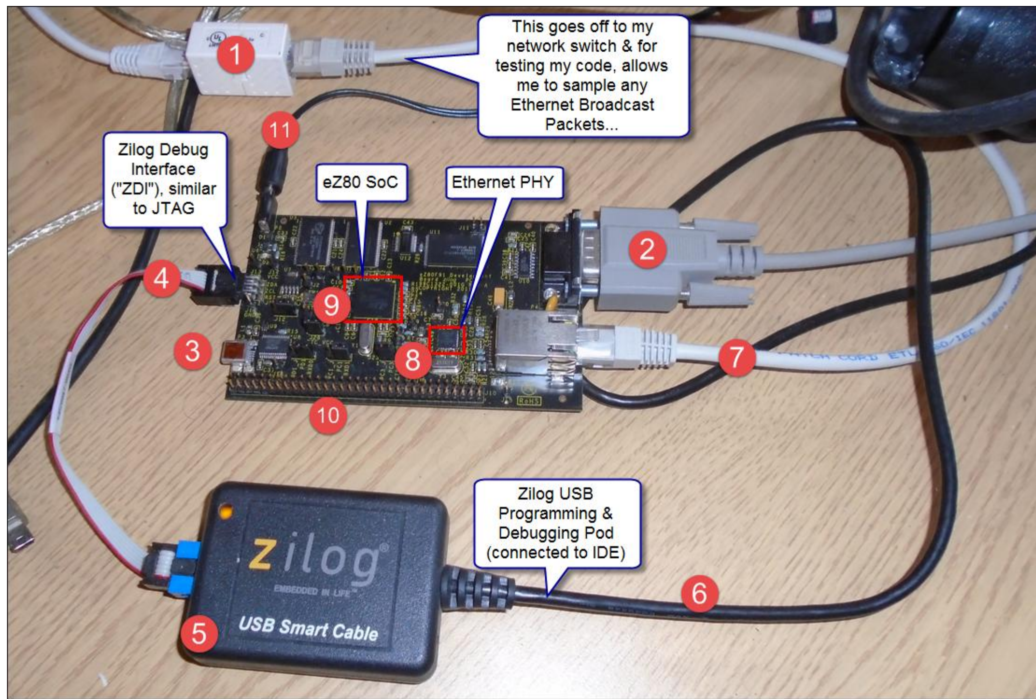
This document and the other referenced *.pdf's are intended to supplement David Husband's CV to showcase/highlight some of his extensive experience, knowledge and abilities...

© 2020 David.Husband@baremetal.engineer, All Rights Reserved Created: 19/08/2020 Updated: 26/11/2020

*All personal information is subject to the new **Data Protection Act 2018** & the **General Data Protection Regulation (EU) 2016/679** ("GDPR")(which remains in force until the end of the transition period on 31 December 2020 & then goes into UK Law) & is used under licence*

[17] However, that would still be very difficult to develop, test and debug without running a terminal program on the RS-232 serial port !!
[18] And the practical work IS ALWAYS "FROM THE BOTTOM UP" - even though the design may be from "the top down"...
[19] For instance, is my USB interface going to be a peripheral or a host?

Appendix A. eZ80 - Development & Testing Set-up



Image# 15 - The Zilog eZ80F91 Development Board Set-up using Zilog's *eZ80F910300KITG* Platform

Like all manufacturers, Zilog supply a development kit for the eZ80 which comprises of quite a substantial system on a circuit board along with an expansion interface on a 0.1" matrix[20]

See Image# 1 for a much bigger view of this board...

This board is quite "resource-rich" by Forth standards[21]...

A naked eZ80[22] has 8k of fast static RAM only, with 256k of slower internal Flash ROM. Internal hardware support is "luxurious" with two UARTs, parallel I/O, I^2C, SPI, Ethernet MAC, Configurable "Chip Selects", "ZBUG" debugging interface with JTAG functionality if needed, low-power standby modes, 32.768KHz driven RTC, etc...

The eZ80 is externally clocked via a 5MHz source and there is an internal PLL multiplier which clocks the eZ80 at a maximum of 50MHz. ***The eZ80 "horsepower" is reckoned to be 80 MIPS @ 50MHz*** compared to the ***4MHz Z80***'s I used in the past which were rated at ***0.58 MIPS @ 4MHz***[23]

If you look at some of the comparative figures in Footnote 22, below, Zilog seem to have engineered the eZ80 well... ****If**** it could be clocked @ 100MHz, it would return 160 MIPS against the ARM Cortex-M3 with 125MIPS @ 100MHz

The current Zilog eZ80 development platform I am using is Zilog's second version...

There is 256k of internal eZ80 flash on both dev boards which defaults at 00-0000 hex to 03-FFFF hex
On the New Dev Board, CS0[24] has 8MB of flash starting @ 14-0000[25] & ending @ 93-FFFF running with 7 wait states
On the New Dev Board, CS1 has 1MB of ram starting @ 0C-0000 & ending @ 13-FFFF with 1 wait state
And another 1MB of ram on CS2, starting @ 04-0000 & ending @ 0B-FFFF with 1 wait state

Referring to Footnote 20 below, Forth itself does not require anything like these resources[26] to run applications well. Referring to Image# 14 and to the "New Model" system I am working towards, the extra Flash would hold the HTML pages that would serve the browser over the Ethernet interface; the ram for the same reason and for data buffering.

---

[20] 0.1" matrix is always a bonus because prototyping hardware is much easier with 0.1" matrix parts, although they are becoming increasingly rare nowadays... Unless you have an extensive prototype surface mount workshop, and production/rework facilities, etc, it is not a practical proposition to perform prototype work at the S.M.D. level... You need S.M.D. skills, too!

[21] I have quite happily run a "Classic" Forth system (see Image# 13) on the basic eZ80 within 8k of internal eZ80 Ram and in about 9k of Flash Rom out of 256k...

[22] But you have to be careful! There are a number of eZ80 versions... I use the fastest, most resourceful version, but it must be understood with these kind of SoC devices, that each pin may well share a number of functions and you must choose which you want -- you cannot have them all!

[23] Intel 8086: 0.33 MIPS @ 5 MHz, Intel 286: 1.28 MIPS @ 12 MHz, NEC V20 (ran 8086 code - I used it!): 4 MIPS @ 8 MHz, Intel i386DX: 2.15 MIPS @ 16 MHz, 4.3 MIPS @ 33 MHz, ARM7: 40 MIPS @ 45 MHz, ARM Cortex-M3: 125 MIPS @ 100 MHz

[24] ***CS0 = Chip Select 0***

[25] I am able to determine the mapping of all these values along with the number of wait states and the type of memory interface as all the ***Chip Selects*** are configurable. This is a ***BIG PLUS-POINT for the eZ80***...

[26] These resources are purely to support the use of Zilog's ANSI C Compiler to run apps in C (They need this much to run!)... And that should tell you a great deal!

So, referring to Image# 15 above, here are some notes against each point highlighted:

(1) I am using this to control the Ethernet packets going to the Dev Board, by plugging and unplugging at this point rather than doing it at Point 7 so as to avoid "wearing" out the socket @ 7

(2) Is the "PC Style" DB9 pin-out serial port which goes to my PC running the "Tera Term" terminal program and linking at 115,200 bauds

(3) This is a "pseudo" RS-232 port over an USB link using one of the FTDI UART-to-USB devices. I don't currently use it

(4) This is the Zilog "ZDI" debugging interface that connects to the Zilog USB Pod (5) supplied with the development kit

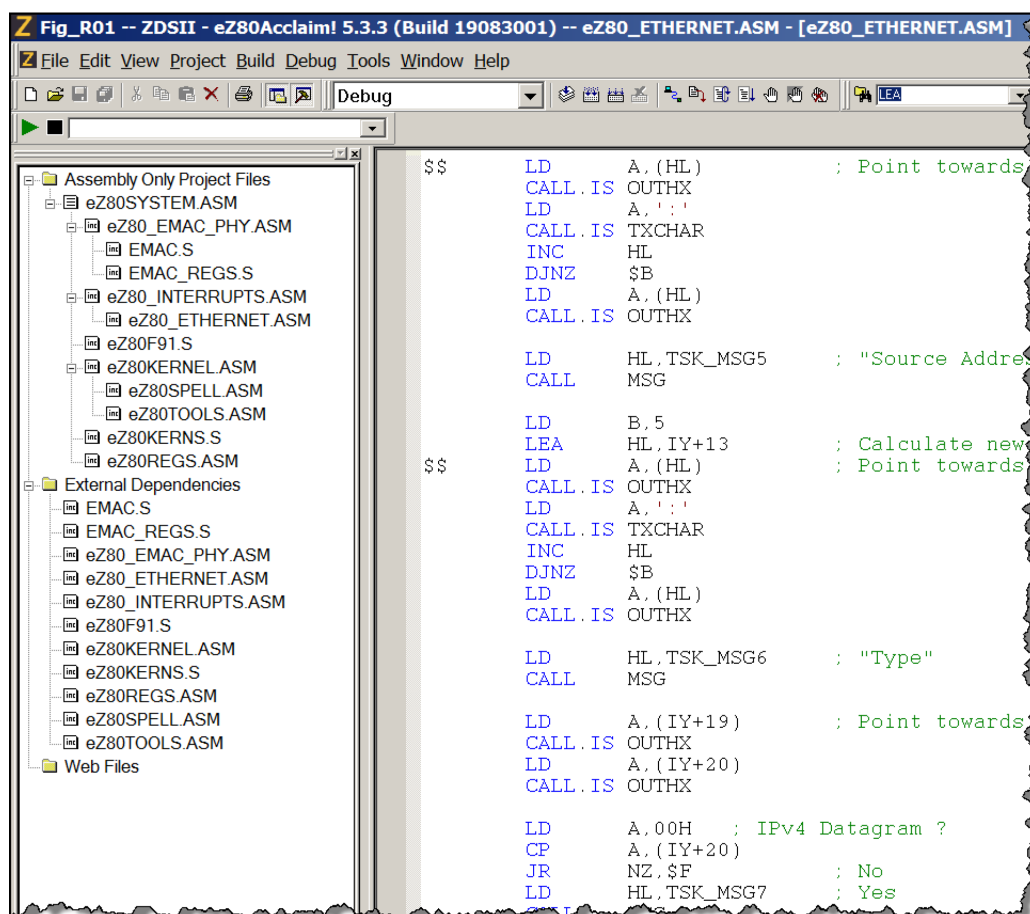
(6) This is the USB lead from the pod that connects to a USB port on my PC and interfaces to the ZDSII Integrated Development Environment ("IDE"), part of which is featured below in Image# 16

(7) Is the Ethernet lead connecting to (1) and then goes on to the Ethernet Port on my PC

(8) & (9) As labelled

(10) Is the expansion bus interface containing a number of useful[27] signal lines...

(11) Is the 5v power in. Alternatively, I could power it via the USB connection at (3)



Image# 16 - A Fragment from Zilog's ZDSII Integrated Development Environment

In Image# 16 above, is a fragment from the ZDSII IDE which Zilog supply free to support the use of their various eZ80 flavours and their ZDI debugging pods. These allow the ZDSII IDE to download and program the internal eZ80 flash and any external flash, and to perform various generic debugging tasks associated with breakpoints and single-stepping...

When you start using these tools from assembler programs, you very quickly become aware that the real intention is for them to support the "C Compiler" which I am sure they do really well...

Also there are a number of serious bugs in the Assembler, and I can see from the Release Notes for the occasions when Zilog performs updates and bug fixes that Zilog's efforts are directed towards fixing bugs that the "C Compiler" throws up. This leads me to believe that most users are using the "C Compiler"... **None of this is a problem for me!!**

[27] And omitting a number of useful signal lines... The J10 connector has 64 pins, but ***all 32*** of the even pins go to 0v -- what a waste when the I^2C lines and a number of other lines (like the pins for the RTC backup) should be on J10...

Appendix B. THE FORTH PARADIGM

Forth was conceived and developed on mainframe computers by Charles Moore and his associates[28] towards the end of the 1960's and throughout the 1970's. During the 1970's Forth was ported to a number of microprocessors by members of the Forth Interest Group (FIG, 2007)

"*Charles Moore is one of the greatest software developers*[29]*. The 'Forth' language he invented is still in use today, particularly by NASA, and has never been bettered for instrumentation and process control. He still argues persuasively that* ***the only way we can develop effective software quickly is to embrace simplicity****. Like Niklaus Wirth, he remains a radical whose views have become increasingly relevant to current software development..*"
(Morris, 2009) (My Emphasis)

> I developed Forth over a period of some years as an interface between me and the computers I programmed. The traditional languages were not providing the power, ease, or flexibility that I wanted. I disregarded much conventional wisdom in order to include exactly the capabilities needed by a productive programmer. The most important of these ***is the ability to add whatever capabilities later become necessary***.

Image# 17 - Extensibility is a major Forth feature

Image: Based on Moore (1981, p. vii)

The Forth language invented in the late 1960's is one of computing's best kept secrets... Widely used but little known….

Forth is highly reflective, which means that most of Forth is written in itself. See "***The Forth Virtual Machine***"

It is a fast, compact, modular, reflective, re-entrant, object-based, extensible, untyped, stack-based, interactive, threaded interpretive, incremental compiler (and operating system); ideally suited to efficient software development and debugging (Frenger, 2001; Simon, 2009) and to driving and controlling all kinds of hardware (Colburn, Moore & Rather, 2011; Pigott, 2006; Simon, 2009)

In fact, the way you "program" an application in Forth is to create word definitions to extend the Forth language set..

Due to the modular and interactive characteristics of Forth, the development methodology is very close to ***Extreme Programming*** ("***XP***")

> One principle that guided the evolution of Forth, and continues to guide its application, is bluntly: ***Keep It Simple*** A simple solution has elegance. It is the result of *exacting effort to understand the real problem* and is recognised by its compelling sense of rightness. I stress this point because it contradicts the conventional view that power increases with complexity. ***Simplicity provides confidence, reliability, compactness, and speed***.

Image# 18 - Charles Moore: Keep it Simple! (Hooker, n.d; Leveson, 1992)

Image: Based on Moore (1981, p. vii)

"***Forth is a design language****. To the student of traditional computer science, this statement is self-contradictory. 'One doesn't design with a language, one implements with a language. Design precedes implementation...' Experienced Forth programmers dis-agree. In Forth you can write abstract, design-level code and still be able to test it at any time by taking advantage of decomposition into lexicons*[30]*. A component can easily be rewritten, as development proceeds, underneath any components that use it. At first the words in a component may print numbers on your terminal instead of controlling stepper motors. They may print their own names just to let you know they've executed. They may do nothing at all. Using this philosophy you can write a simple but testable version of your application, then successively change and refine it until you reach your goal*"
(Brodie, 2004, p. 31)

---

[28] Elizabeth Rather, Leo Brodie, Kim Harris...

[29] Also rather a clever hardware engineer (now a multimillionaire) who holds very valuable patents on a number of fundamental microprocessor innovations (MMPP, 2007). See: ***http://spectrum.ieee.org/at-work/innovation/qa-with-moores-ip-manager*** or Google "***Moore Microprocessor Patent Portfolio***"

[30] One meaning of lexicon is "a set of words pertaining to a particular field of interest" (***http://www.thefreedictionary.com/lexicon***)

Develop from a Prototype (with little planning)

Moore and Brodie take their approach further so that the various processes become a method of ***problem-oriented solution thinking***... In essence:

> **"Get a bare-bones application running quickly. Demonstrate it and get feedback from users. Then modify and expand capability: much more satisfactory than planning in advance"**
> **[Moore]**

Image# 19 - Develop from a Prototype

Image: Based on (Morris, 2009)

There are some advantages in this approach – you very quickly create a working application prototype of some kind to show your stakeholders and then you can develop this prototype[31]… Run on prototype hardware too, until/while "real" hardware platform is being designed/developed… However, the analysis of the problem and the initial starting design must be sound...

The Forth Modus Operandi

The Basic Idea

Forth is expressed in words and numbers and is separated by spaces, i.e.:

HAND OPEN ARM LOWER HAND CLOSE ARM RAISE

These commands may be typed directly from the keyboard or edited onto mass storage and loaded

All words, whether included with the system or user-defined, exist in the "dictionary", a singly linked list

A "***defining word***" is used to add new words to the dictionary. One defining word is **:** (pronounced "***colon***"), which is used to define a new word in terms of previously defined words. Here is how one might define a new word called ***LIFT***

: LIFT HAND OPEN ARM LOWER HAND CLOSE ARM RAISE ;

The final **;** terminates the definition. The new word **LIFT** may now be used instead of the long sequence of words that comprise its definition

Forth words can be nested like this indefinitely[32] and writing a Forth application consists of building increasingly powerful definitions, such as this one, in terms of previously defined words

Implicit Calls

To execute (or "run" or invoke) the word **LIFT** for instance, you don't have to say **CALL LIFT** you just type **LIFT** or it is encountered in the input stream and is invoked

Implicit Data Passing

Passing data in Forth is implicit and is achieved via Forth's Parameter Stack, which in most implementations is the microprocessor's stack. However, there are no **PUSH** or **POP** operations in high-level Forth

The implications of Implicit Calling and Implicit Data Passing

As data is passed implicitly, ***we are relieved of the act of passing data to and from our code***, leaving us ***to concentrate upon the functional steps of the data's transformation***

Passing data via a stack has the advantage that words can nest within words, because any word can put numbers on the stack and take them off without upsetting the flow of data between words at a higher stack level[33]. In this way, the stack supports structured, modular programming while providing a simple mechanism for passing local arguments

Forth eliminates from our programs the details of how words are invoked and how data is passed
What does that leave? ***Only the words that <u>describe our problem</u>...***

Having words, we can fully exploit the recommendations of Parnas (1972):

---

[31] Contrary to other development methods where the initial prototype may/will be discarded

[32] And Forth's ***Return Stack*** automagically keeps track of this nesting...

[33] Provided that the word doesn't consume or leave any unexpected values...

"*to decompose problems according to things that may change, and to have each module consist of many small functions, as many as are needed to hide information about that module*"

Forth allows us to write as many words as we need, no matter how simple they may be

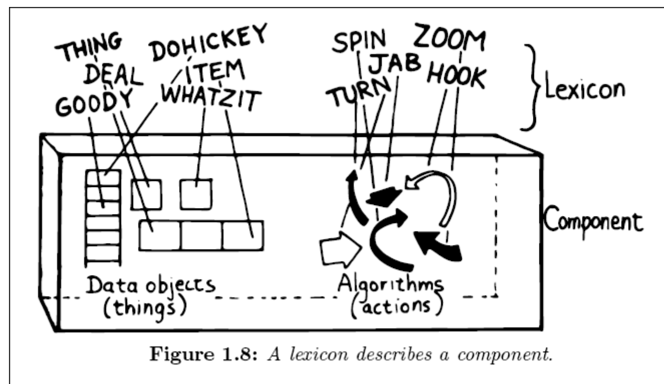
Programming with Components

Having a large set of simpler words makes it easy to use a technique that Brodie calls "***component programming***"

He defines a component as "*the smallest set of interacting data structures and algorithms that share knowledge about how they collectively work...*" (Brodie, 2004, p. 20)
In reality, they are just a collection of well-chosen and well-designed Forth words...

A component represents a resource which can be a piece of hardware such as a UART, or a software resource such as a queue or an object, and all components will involve data objects and algorithms

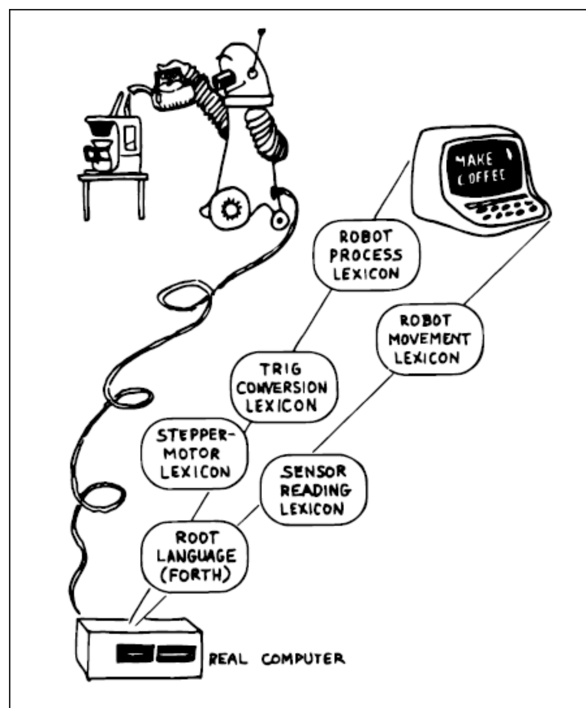
Brodie calls the Forth words that make up the component, the "Lexicon". The design of the lexicon is very important as the essence of a Forth application is the creation of the appropriate problem-solving set of words as an extension to the core set of Forth words



Image# 20 - An example Forth Application's Lexicon & associated Component

Image: Based on (Brodie, 2004, p. 22)

Forth is word-based, so a real Forth application consists of a number of words all working together to provide a functionality set. These words represent the various components defined, identified and documented during the analysis and design process... See Image# 21 below

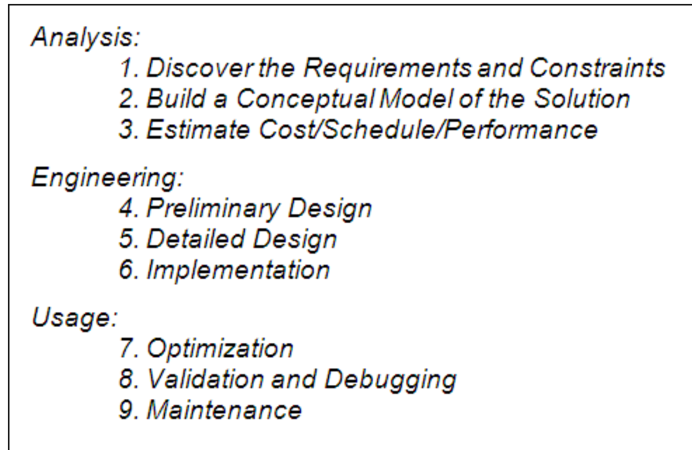


Image# 21 - The Entire Application Consists of Components represented by Lexicons

Image: (Brodie, 2004, p. 23)

Problem-oriented Solution Thinking

Brodie suggests nine phases to this problem-oriented solution thinking activity:



Analysis:
1. *Discover the Requirements and Constraints*
2. *Build a Conceptual Model of the Solution*
3. *Estimate Cost/Schedule/Performance*

Engineering:

4. *Preliminary Design*
5. *Detailed Design*
6. *Implementation*

Usage:

7. *Optimization*
8. *Validation and Debugging*
9. *Maintenance*

Image# 22 - The Nine Development Phases

Image: Based on (Brodie, 2004, p. 38)

Iteration - The Scientific Method

This is what drives all the efforts behind the project and as described by Harris (1981) is based upon the scientific method, which is itself iterative, being ...

> ... a never-ending cycle of discovery and refinement. It first studies a natural system and gathers observations about its behavior. Then the observations are modeled to produce a theory about the natural system. Next, analysis tools are applied to the model, which produces predictions about the real system's behavior. Experiments are devised to compare actual behavior to the predicted behavior. The natural system is again studied, and the model is revised.
>
> The *goal* of the method is to produce a model which accurately predicts all observable behavior of the natural system.

Image# 23 - The Scientific Method.

Image: Based on Brodie (2004, p. 39) citing Harris (1981)

The Iterative Approach to Development

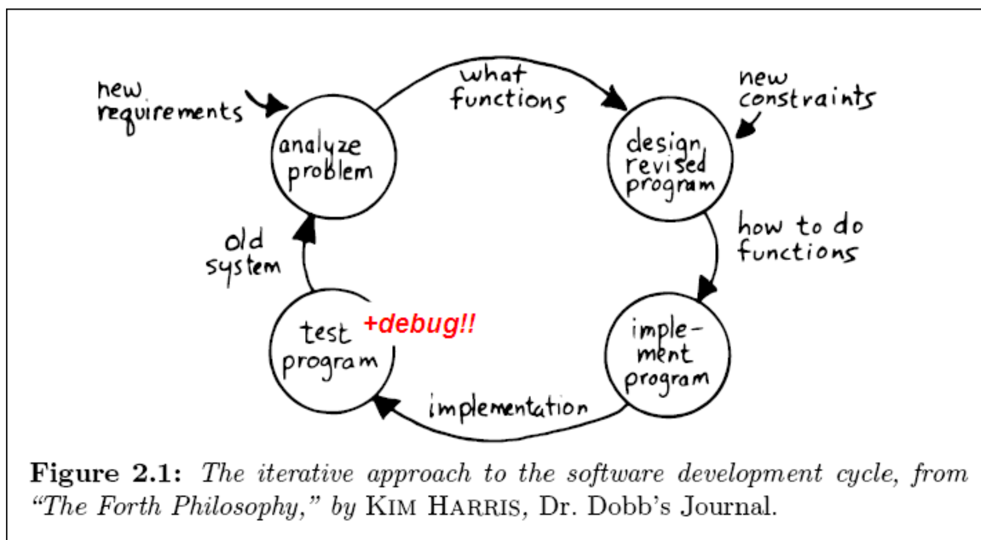


Figure 2.1: *The iterative approach to the software development cycle, from "The Forth Philosophy," by* Kim Harris, Dr. Dobb's Journal.

Image# 24 - An Iterative Approach to Development

Image: Husband, 2011, modified[34] from Brodie (2004, p. 39) citing Harris (1981)

[34] ... by adding "+*debug*" to "*Test Program*" stage... That was a serious omission....!!

The Iterative Approach to Analysis

Referring to Image# 24 above, Brodie breaks down the "***Analyse Problem***" phase into an-other iterative cycle, shown below in Image# 25

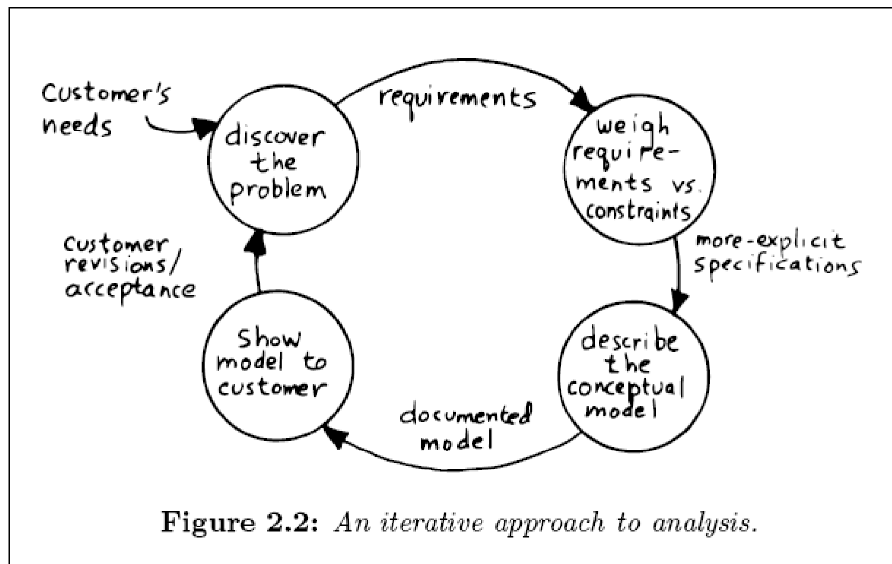


Figure 2.2: *An iterative approach to analysis.*

Image# 25 - An Iterative Approach to Analysis.

Image: Brodie (2004, p. 48) citing Harris (1981)

Start with the Simplest Solution & Few Constraints

> Software development in Forth seeks first to find the simplest solution to a given problem. This is done by implementing selected parts of the problem separately and by ignoring as many constraints as possible. Then one or a few constraints are imposed and the program is modified...

Image# 26 - Develop with very few initial constraints

Image: Brodie (2004, p. 40) citing (Harris, 1981)

The Importance of the Conceptual Model of the proposed Solution

It must be self-evident that if the conceptual model of the solution is incorrect and/or deficient then the whole project may not be viable...

The Conceptual Model *is* Forth...

As far as the software is concerned, the application is not "written in Forth"; Forth is the application. The language is extended as required, to contain word sets ("Lexicons") which describe and implement the chosen functions and solutions

Some Tips when Developing the Conceptual Model:

- *Strive to build a solid conceptual model before beginning the design*
- *First, and most importantly, the conceptual model should describe the system's interfaces*
- *Decide on error- and exception-handling early as part of defining the interface*
- *Develop the conceptual model by imagining the data travelling through and being acted upon by the parts of the model*
- *You don't understand a problem until you can simplify it*
- *Generality usually involves complexity. Don't generalize your solution any more than will be required; instead, keep it changeable*
- *To simplify, quantize*
- *To simplify, keep the user out of trouble*
- *To simplify, take advantage of what's available*

Image# 27 - Tips for Developing the Conceptual Model

Image: All points from Brodie (2004, pp. 48-65)

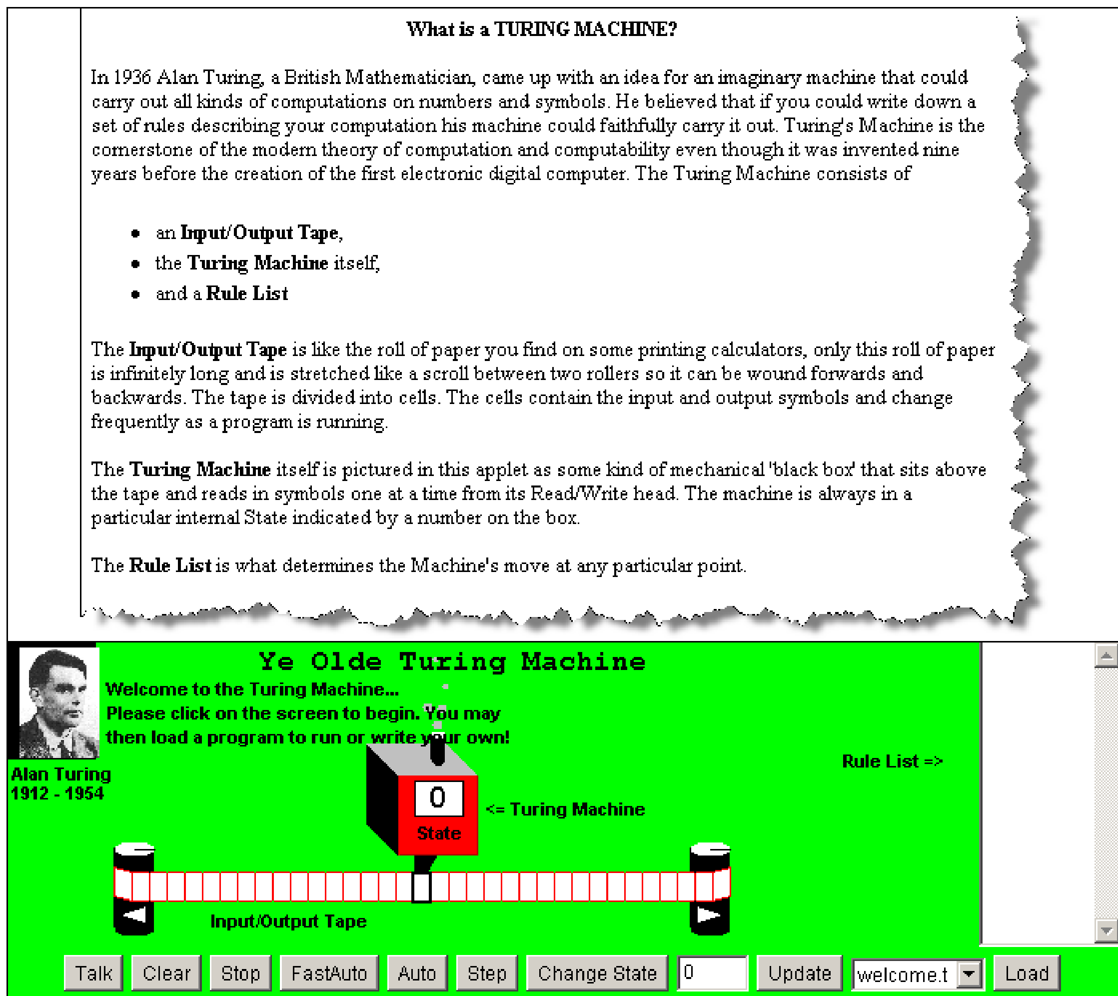
Appendix C. THE FORTH VIRTUAL MACHINE

What is a "Virtual Machine" ("VM") ?

It is *"an 'abstract' computing architecture or computational engine that is independent of any particular hardware or operating system…."* (Taivalsaari, 2003)

A virtual machine can be extremely powerful. It need only exist in somebody's mind…

The classic VM is the product of one of computing science's greatest minds; Alan Turing's 1936 ***thought experiment***, now known as the "***Turing Machine***". (Turing, 1936)



What is a TURING MACHINE?

In 1936 Alan Turing, a British Mathematician, came up with an idea for an imaginary machine that could carry out all kinds of computations on numbers and symbols. He believed that if you could write down a set of rules describing your computation his machine could faithfully carry it out. Turing's Machine is the cornerstone of the modern theory of computation and computability even though it was invented nine years before the creation of the first electronic digital computer. The Turing Machine consists of

- an **Input/Output Tape**,
- the **Turing Machine** itself,
- and a **Rule List**

The **Input/Output Tape** is like the roll of paper you find on some printing calculators, only this roll of paper is infinitely long and is stretched like a scroll between two rollers so it can be wound forwards and backwards. The tape is divided into cells. The cells contain the input and output symbols and change frequently as a program is running.

The **Turing Machine** itself is pictured in this applet as some kind of mechanical 'black box' that sits above the tape and reads in symbols one at a time from its Read/Write head. The machine is always in a particular internal State indicated by a number on the box.

The **Rule List** is what determines the Machine's move at any particular point.

Image# 28 - A modern online Java representation/implementation of a Turing Machine

Image: Husband, 2011, based on: Schweller (2003)

Moving forward in time to 1968, legendary computer scientist, Donald Knuth published the seminal series of books called "***The Art of Computer Programming***"

His work featured an imaginary virtual machine called "***MIX***" (See Image# 29 below) and its accompanying MIX assembly language[35]

Subsequently a number of high-level languages have employed a VM internally to avoid platform dependence and to isolate programs from hardware details. There are other reasons... See Image# 30 below

As Taivalsaari points out in Image# 31 below, the Forth VM is very simple[36]; I would characterise it more as a "Virtual Microprocessor"….

[35] A man who was not afraid to use assembly language...

[36] Surprise! Surprise!

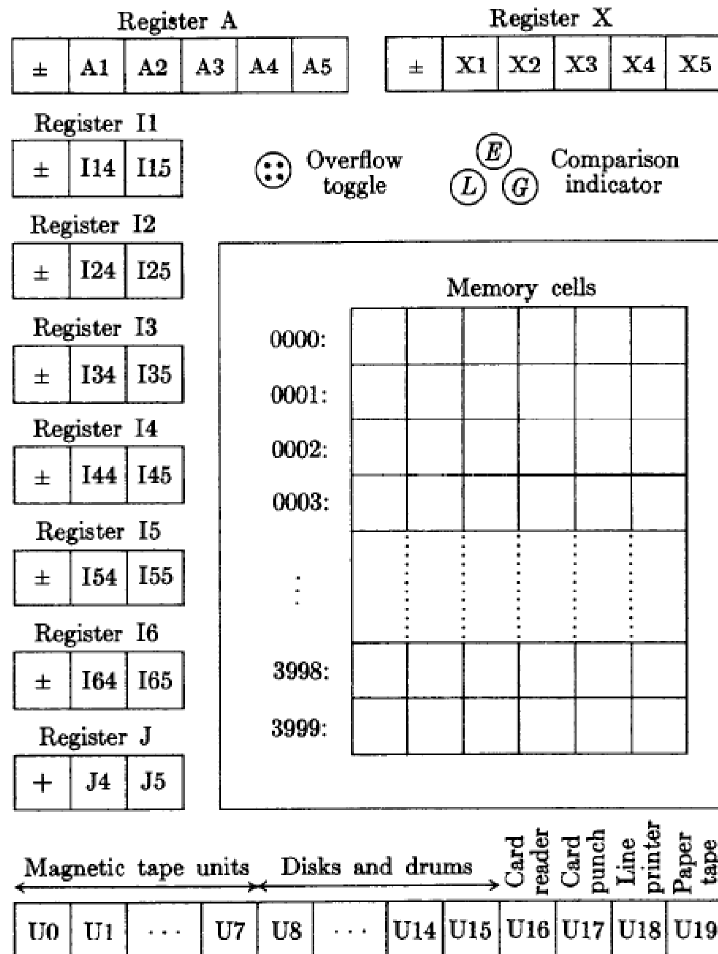


Fig. 13. The MIX computer.

Image# 29 - Knuth's MIX Virtual Machine

Image: (Knuth, 1997, p. 126)

Languages that Use Virtual Machines

- Well-known languages using a virtual machine:
  - *Lisp* systems, 1958/1960-1980s
  - *Basic, 1964-1980s*
  - *Forth*, early 1970s
  - *Pascal* (P-Code versions), late 1970s/early 1980s
  - *Smalltalk*, 1970s-1980s
  - *Self*, late 1980/early 1990s
  - *Java*, mid-1990s
- Numerous other languages:
  - ... *PostScript*, *TCL/TK*, *Perl*, *Python*, *C#*, ...

Image# 30 - Languages that use Virtual Machines

Image: (Taivalsaari, 2003, p. 13)

Why is Forth Interesting from the VM Designer's Viewpoint?

- One of the easiest virtual machines to build.
- The VM consists of a small number of distinct components (stacks, dictionary, interpreter, virtual registers, primitives); no extra “fat”.
- The language itself is small, simple and efficient, and provides an unusual combination of high-level abstraction and very low level programming capabilities.
- High level of reflection (significant portions of the VM written in the language itself.)
- Ideal for embedded systems (if the awkward syntax is not exposed to the end user...)

Image# 31 - Forth is an “interesting” VM...

Image: (Taivalsaari, 2003, p. 30)

In Image# 31 above, Taivalsaari's last comment is: "*Ideal for embedded systems (if the awkward syntax*[37] *is not exposed to the end user...*" I do not agree with his personal opinion...

He is from Finland[38]. He clearly did not like (or take to) Reverse Polish! (as he says the Finnish equivalent of "Red House", so reverse polish will instinctively feel strange to him)..
Footnote 36 repeats what I said in Footnote 13 earlier...

The Forth Virtual Machine Registers

A virtual machine must have virtual machine registers to hold the data that it works with...
figForth’s are shown Image# 32 below, along with how they are mapped to real eZ80 registers

	figForth VM	*Z80*
SP	Data stack pointer	*SP*
RP	Return stack pointer	*(memory)*
IP	Interpretive pointer	*BC*
W	Current word pointer	*DE*

Image# 32 - Forth VM Registers mapped to eZ80 Registers (Typical mapping)

Image: Husband, 2011, based upon (Ting, 1989, p. 27)

Inner Interpreters (a.k.a. "Address Interpreters")

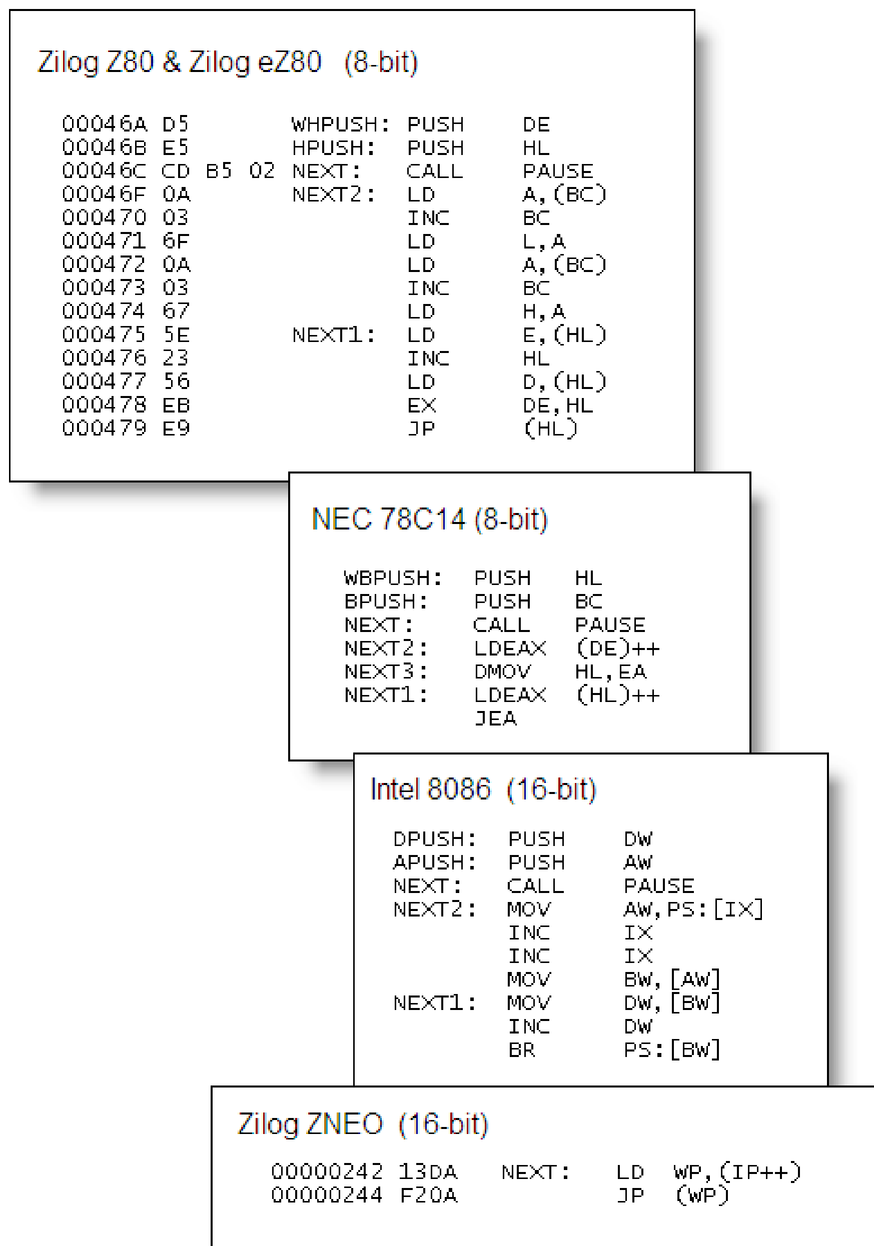
Ting (1986, p. 52) takes a wide (and very insightful) view of Forth inner interpreters, saying that they “*are a set of execution procedures, usually in the machine code of the host computer, which execute various Forth words by processing the information stored in their parameter fields. The address of such a procedure is stored in the code field of a word definition. Forth definitions of the same class have the same address in their code fields. Two major inner interpreters are used to process code definitions defined by machine instructions and colon definitions defined in terms of other existing Forth words…*”

I am not convinced that there are two code interpreters as Ting claims;
I only see one and I see the other as a (re-)entry point to the address interpreter...

---

[37] In English we say "***Red House***"; in French they say "***Maison Rouge***" -- "***House Red***".... Reverse Polish Notation is quite a straightforward way of operating and makes quite a few things much easier... Remember: ***Complexity or Simplicity?***

[38] In English we say "***Red House***"; in Finnish they say "***Punainen Talo***" ("Red House"), where "House" = "Talo" and "Red" = "Punainen". I rest my case... Q.E.D.



Image# 33 - Some Inner Interpreters (Address Interpreters) implemented on various microprocessors

Image: Husband, 2011, based upon some work done since 1983 based on the *fig*Forth models. Zilog ZNEO based upon work by Rodriguez (2006)

A number of “*other minor inner interpreters are used to process constants, variables, user variables and other types of data and structures*” (Ting, 1986, p. 52)

I like the way Ting perceives these Forth words as separate Interpreters (and Compilers) because that has enhanced my own understanding of them...

Appendix D. MULTI-TASKING (TIME MULTIPLEXING)

The Role of Multitasking in an Event-Driven Architecture

Multi-tasking is ***vital to implementing an event-driven architecture*** as it allows each event to have its own task or task handler and to execute as if it were an independent module or thread...
Multi-tasking generates the illusion of concurrency

What is "Multi-tasking"?

Wicklund says "*a task is a software construct defining a segment of code that runs as an independent process or function...*" (Wicklund, 1982), so multitasking is where a number of tasks are executed in turn, rapidly enough so as to appear to be running concurrently (or "in parallel")

In this project I have pursued "***closed***" multi-tasking, where the only tasks are control tasks which is ideally suited to an embedded system

Forth as originally devised by Charles Moore was multi-tasking, ran on a mainframe and supported a number of users on the end of remote terminals. When it eventually morphed into figForth, the multi-user stuff was taken out, but its original structure and architecture was not changed... This means that figForth is easy to multi-task[39]

Combining Pre-emptive with Co-operative multitasking

Multitasking modes do not have to be "either-or"; they can be combined very effectively. In Fig 204 is some legacy Z80 code I wrote in 1988 to implement a simple pre-emptive multitasker on an interrupt being triggered at 60Hz by a Clock/Calendar IC

This interrupt is too often (for efficiency) and so is divided by 12; so five times per second each subroutine in the table SCHTAB is executed. This means that each subroutine is executed once per second. Four of them are dummies for use later if needed. The first entry is a clock task which only sets a flag[40]

```
NMIROT: PUSH    AF
        PUSH    BC
        PUSH    DE
        PUSH    HL

        LD      HL,NMICNT
        DEC     (HL)

        LD      A,(HL)
        OR      A
        JP      NZ,NMIRO2       ; skip if not zero

        LD      A,12
        LD      (HL),A

        LD      A,(SCHCNT)
        PUSH    AF
        ADD     A,A
        LD      HL,SCHTAB
        LD      DE,0
        LD      E,A
        ADD     HL,DE
        POP     AF
        INC     A
        CP      5
        JR      NZ,NMIRO3
        LD      A,0
NMIRO3: LD      (SCHCNT),A
        JP      (HL)

NMIRO2: POP     HL
        POP     DE
        POP     BC
        POP     AF
        RETN

SCHTAB: JR      SCH1    ; CLOCK TASK
        JR      SCH2
        JR      SCH3
        JR      SCH4
        JR      SCH5

SCH1:   LD      HL,TASKS
        SET     0,(HL)          ; CLOCK TASK
        JP      NMIRO2

SCH2:   JP      NMIRO2

SCH3:   JP      NMIRO2

SCH4:   JP      NMIRO2

SCH5:   JP      NMIRO2
```

The Non-Maskable Interrupt ("NMI") is being invoked 60 times per second by a Clock-Calendar IC
.... this is a bit too often for our purposes, so a counter is used... and decremented on each interrupt...
... exit if the count is not zero...
counter is zero, so reload the count...
We arrive at this point, 5 times per second (60/12)...
.... we cycle through the entries in SCHTAB, executing each one in turn... SCHCNT remembers where we are in SCHTAB for the next time through...
This task is executed once per second and runs the internal time count... (clock)
Interrupt routines should do as little processing as possible in the interrupt path. This is because in this instance, the use of the NMI makes this interrupt path "atomic"; i.e. nothing can stop it running until it exits, so it brings the foreground system to a complete halt until it is done. If this went on for any length of time, the foreground system will become obviously unresponsive. When I use maskable interrupts, I am in the habit of disabling any further interrupts so these routines are also atomic...
Important: all it does in the interrupt path is to set a flag then exit...
... these task slots are all "spare" so are "stubbed-out" and merely exit the interrupt routine.... they can be used if required for monitoring switches or led's, etc...

Image# 34 - Pre-Emptive Multitasking by using an Interrupt Routine

Image: Husband, 2011 based upon work done in 1988

[39] If you know how...
[40] This flag just happens to trigger a ***co-operative task*** which performs the real work but only when other programs permit it to…

Simple "closed" multitasking to allow Objects/Code to "Background Execute"

```
PAUSE:  DI
        PUSH    AF
        PUSH    HL
        LD      HL,TASKS
        BIT     7,(HL)
        JR      NZ,PAUS1
        BIT     0,(HL)
        CALL    NZ,CTASK
        BIT     1,(HL)
        CALL    NZ,PTASK
        BIT     2,(HL)
        CALL    NZ,ATASK
        BIT     3,(HL)
        JR      NZ,ESCAPE
        BIT     4,(HL)
        CALL    NZ,GTASK
        BIT     5,(HL)
        CALL    NZ,OTASK
        BIT     6,(HL)
        CALL    NZ,PRITSK
PAUS1:  POP     HL
        POP     AF
        EI
        RET

CTASK:  SET     7,(HL)
        EXX
        EX      AF,AF'
        LD      HL,TASKS
        RES     0,(HL)
        LD      BC,CLOCKT
        JP      NEXT2

PTASK:  SET     7,(HL)
        JP      NEXT2

ATASK:  SET     7,(HL)
        JP      NEXT2

ESCAPE: RES     3,(HL)
        RES     7,(HL)
        CALL    INTSET
        EI
        LD      BC,QUITP
        JP      NEXT2

GTASK:  SET     7,(HL)
        EXX
        EX      AF,AF'
        LD      HL,TASKS
        RES     4,(HL)
        LD      BC,(INTCHR)
        JP      NEXT2

OTASK:  SET     7,(HL)
        JP      NEXT2

PRITSK: SET     7,(HL)
        JP      NEXT2
```

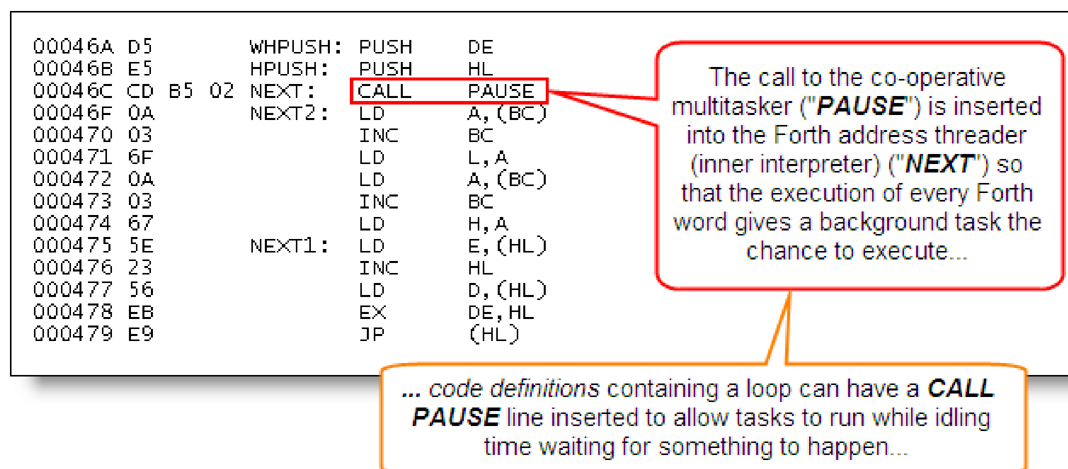
```
TASKS EQU CBYTE-1        ; BIT 0 CLOCK TASK
                         ; BIT 1
                         ; BIT 2
                         ; BIT 3 ESCAPE
                         ; BIT 4 GENERAL TASK
                         ; BIT 5
                         ; BIT 6
                         ; BIT 7 TASK RUNNING FLAG
```

```
CLOCKT: WORD    $+2
        WORD    DOCOL                ; :
        WORD    TICKS                ; TICKS
        WORD    LIT,24,EQUAL         ; 24 =
        WORD    ZBRAN,CLOCT1-$       ; IF
        WORD    ZERO,ZERO            ; 0
        WORD    XCLOC,DSTOR          ; CLOCK D!
CLOCT1:                              ; THEN
        WORD    DROP,DROP            ; DROP DROP
        WORD    LIT,CLOCK,DAT        ; CLOCK D@
        WORD    ONE,ZERO,DPLUS       ; 1 D+
        WORD    LIT,CLOCK,DSTOR      ; CLOCK D!
        WORD    RETURN               ; RETURN
```

```
RETURN: WORD    $+2
        LD      HL,(RPP)
        INC     HL
        INC     HL
        LD      (RPP),HL
        EX      AF,AF'
        EXX
        RES     7,(HL)
        RET
```


Image# 35 - Co-operative Multitasking by invoking a simple scheduler

Image: Husband, 2011 based upon work done in 1989



```
00046A D5            WHPUSH: PUSH   DE
00046B E5            HPUSH:  PUSH   HL
00046C CD B5 02      NEXT:   CALL   PAUSE
00046F 0A            NEXT2:  LD     A,(BC)
000470 03                    INC    BC
000471 6F                    LD     L,A
000472 0A                    LD     A,(BC)
000473 03                    INC    BC
000474 67                    LD     H,A
000475 5E            NEXT1:  LD     E,(HL)
000476 23                    INC    HL
000477 56                    LD     D,(HL)
000478 EB                    EX     DE,HL
000479 E9                    JP     (HL)
```


Image# 36 - Inserting a call to the multitasker into a high-level loop

Image: Husband, 2011 based upon work done in 1988. Z80 code based on the Intel 8080 *fig*Forth model

And this is from my recent work in ADL[41] mode on the eZ80[42], based upon my previous work as shown in Image# 35 & Image# 36 above..

```
0002AF F3          PAUSE:  DI
0002B0 F5                  PUSH  AF
0002B1 C5                  PUSH  BC
0002B2 D5                  PUSH  DE
0002B3 E5                  PUSH  HL
0002B4 DDE5                PUSH  IX
0002B6 FDE5                PUSH  IY

0002B8 21F7FF              LD    HL,TASKS
0002BB CB7E                BIT   7,(HL)
0002BD 20 1C               JR    NZ,PAUSE1    ; Skip if task running

0002BF 21F7FF              LD    HL,TASKS
0002C2 CB46                BIT   0,(HL)
0002C4 52C4 4A 04 01       CALL.IL NZ,TASK0   ; Ethernet Packet Task
                                              ; (In

0002C9 21F7FF              LD    HL,TASKS
0002CC CB4E                BIT   1,(HL)
0002CE 52C4 7C 0B 01       CALL.IL NZ,ARP_TASK  ; ARP Cache Freshness Task
                                                ; (In eZ80_ETHERNET.ASM)

0002D3 21F7FF              LD    HL,TASKS
0002D6 CB5E                BIT   3,(HL)
0002D8 C2 79 02            JP    NZ,ESC_KEY

0002DB FDE1       PAUSE1:  POP   IY
0002DD DDE1                POP   IX
0002DF E1                  POP   HL
0002E0 D1                  POP   DE
0002E1 C1                  POP   BC
0002E2 F1                  POP   AF
0002E3 FB                  EI
0002E4 C9                  RET
```

Disable Interrupts

Save all the registers

Fetch the Task Control Byte called "TASKS"

If Bit 7 is set, there is a task already running, so exit.. Otherwise "ripple through" to test Bit 0..

Because BIT 0 is tested 1st, it is the "top priority" task Bit 0 is set from the Ethernet Rx Pkt interrupt, so TASK0 is called with a "long-call"

Now test Bit 1, which is set from within DO_CACHE in TASK0 ARP_TASK is called with a "long-call" Otherwise "ripple through" to test Bit 3..

Finally, test Bit 3, which is set by testing the input stream from the serial port for 1B hex which is the escape key code

The response to Bit 3 being set is to make a JUMP to the ESC-KEY Task which does some housekeeping then makes the Forth system perform a warm-start..

More on the ESC-KEY Task in further images...

Finally, restore all the saved registers, re-enable the interrupts & return to the callee..

I devised this method about 30 years ago with no external input..
It is fast, simple, small, versatile & VERY ROBUST...
This is part of the core of an EVENT-DRIVEN SYSTEM !!
(And this is why I describe myself as an "Intuitive Engineer" !!)

Image# 37 - PAUSE - a simple, but very effective, co-operative multi-tasking mechanism...

When ESC-KEY is entered by jumping out of PAUSE, HL is still pointing to TASKS & there is a stackfull of preserved registers from PAUSE plus interrupts are still disabled..

... plus, two of the Task Control bits are still set... So clear those. The circular interrupt buffer for the serial port to/from the PC Terminal Program is reset by resetting the "Putting-In pointer" & the "Taking-out pointer" Same for the Ethernet Packet buffer filled by the Ethernet Rx Pkt interrupt...

```
000279           ESC_KEY:
000279 CB9E                RES   3,(HL)
00027B CBBE                RES   7,(HL)
00027D 217FFF              LD    HL,CBUFF1      ; Initialise Console
000280 22E7FF              LD    (CB_PTR1),HL  ; buffer pointers
000283 22E5FF              LD    (CB_PTR2),HL

000286 5B21 00 00 0C       LD.LIL HL,PktBuffer
00028B 5B22 FA FF 0B       LD.LIL (PktBufInPtr),HL   ; Initialise Packet
000290 5B22 FD FF 0B       LD.LIL (PktBufOutPt

000295 FB                  EI
000296 01 64 02            LD    BC,QUITP
000299 C3 31 09            JP    NEXT2
```

QUITP is the address of a custom Forth word loaded into BC which is the Forth VM's IP A JUMP to NEXT2 enters Forth's VM (a.k.a. "Inner Interpreter" a.k.a. "Address Threader") From here on, control is handed to the Forth System..

This is the software equivalent to "kick-starting" a motorcycle engine !! Because it is "kick-starting" the Forth System by entering the Forth VM & warm-starting from pure assembler...

Image# 38 - How the ESC-KEY Task transitions from pure assembler to the Forth VM. Part of the execution behaviour of pressing the "Esc" key on the terminal is determined by ESC_KEY

[41] ADL= "Address & Data Long" mode = 24-bit extended addressing and 24-bit extended data mode...

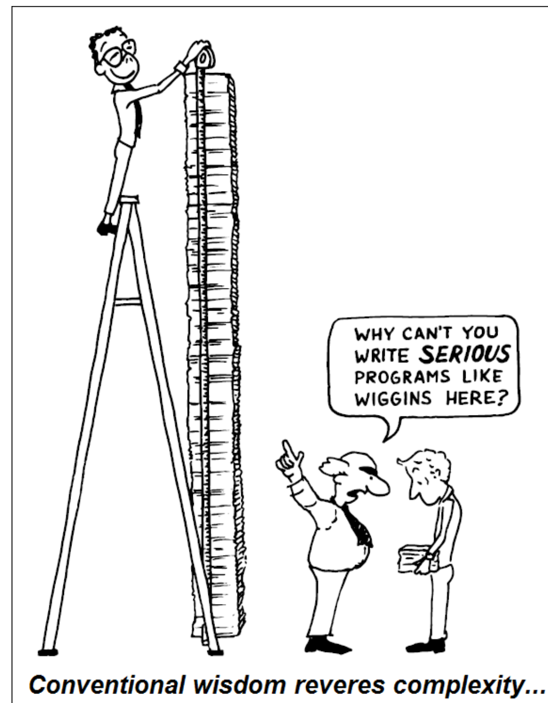
[42] As described in the first 12 pages of this document...

So finally the move from pure assembler (via ESC-KEY) to entering the Forth VM...
QUITP is executed by the Forth "Inner Interpreter" or "address threader" **NEXT**
& leads to a Forth System warm-start via the Forth Outer Interpreter (Forth's
"SHELL") **QUIT**

```
000264 6602      QUITP: DW    $+2
000266 8D10             DW    DOCOL
000268 2B0D             DW    SPSTO
00026A F936             DW    DEC
00026C 2138             DW    PDOTQ
00026E 080D0A45 73636170 DB  8,ACR,LF,"Escape"
000276 65
000277 3A02             DW    QUIT
```

Image# 39 - How pure assembler, via ESC-KEY, transitions into the Forth VM via QUITP, which determines the final execution behaviour of the "ESC" key with Forth words...

Appendix E. Complexity or Simplicity?



Conventional wisdom reveres complexity...

Image# 40 - Resist the Pressures - Reject Complexity...

Image: Based upon (Brodie, 1981, p. 67)

Charles Moore[43] (the inventor of Forth) argues that the only way to develop & test effective software quickly is to embrace simplicity...

> **"I despair. Technology, and our very civilization, will get more and more complex until it collapses. There is no opposing pressure to limit this growth."** [Moore]

Image# 41 - The Complexity Crunch

Image: Husband, 2011, based on Moore cited by Morris (2009)

Moore's sentiments are echoed by the inventor of Pascal, Niklaus Wirth[44] (1971) in a seminal paper, citing Reiser: ***"Software is getting slower more rapidly than hardware becomes faster..."***

> It should be recognized that the single most important contribution towards a design's reliability is *the elimination of superfluous features and facilities, and the containment of its complexity.*
>
> —*Niklaus Wirth*

Image# 42 - Eliminate Complexity & Superfluous Features

Image: Based on (Aguilar, 1999) citing Wirth

Moore's problem approach philosophy, developed in the 1960's & reflected in Forth, echoes the modern fashion for highly iterative ***Agile & Extreme Programming*** methods as a software development process (Frenger, 2001)

A further complication is that of a closed system's disorder to increase over time. This is known as "*entropy*" and was applied to software by Jacobson et al. (1992, pp. 69-70).

---

[43] It would be a fair comment to say that Moore has a very well-founded "*obsession*" with the need to reduce and manage complexity. This approach is echoed by Wirth (1971), Leveson (1995) & Flynt (2004) who devotes a large part of his book to addressing the many complexity issues that arise during object-oriented software engineering... (And none of them are "Forth" people...)

[44] Wirth's Law - ***http://en.wikipedia.org/wiki/Wirth%27s_law***

Appendix F. An Event-Driven Environment

Embedded systems by their nature are attempting to model some clearly-defined aspect(s) of real-world behaviour[45] which in turn could be characterised as being asynchronous and event-driven.

An event-driven architecture offers a number of advantages:

- Separation of Concerns[46] – See Image# 43 below
- Event Processing logic can be separated from the application making it easier to extend or modify
- Changes in state can be responded to as they occur
- Event-based systems may be easier to scale
- When an event-driven system is coupled to message-oriented middleware (MOM)[47], the systems can be geographically separated

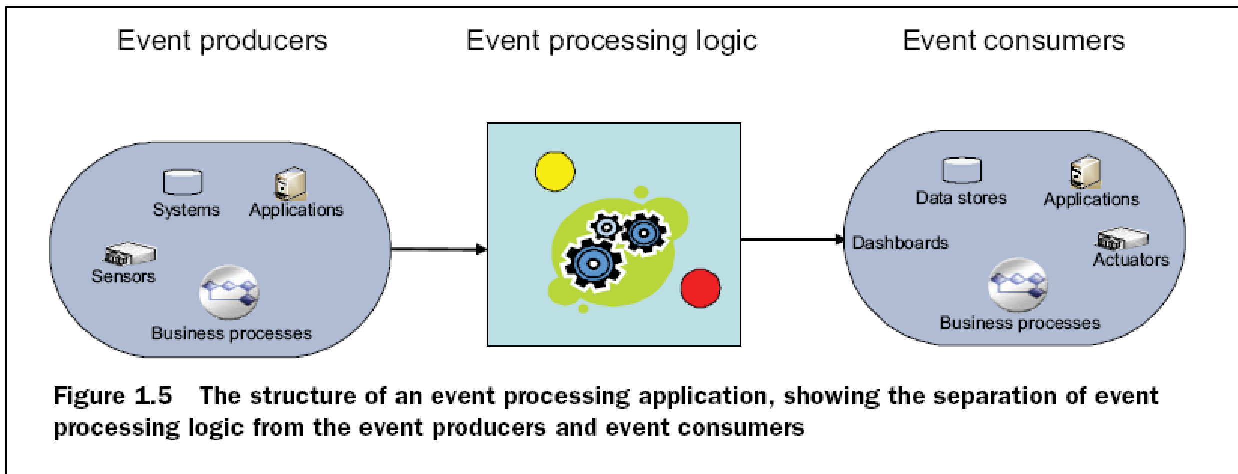


Figure 1.5 The structure of an event processing application, showing the separation of event processing logic from the event producers and event consumers

Image# 43 - The Structure of an Event Processing Application.

Image: (Etzion & Niblett, 2011, p. 14)

EVENT An *event* is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain. The word *event* is also used to mean a programming entity that represents such an occurrence in a computing system.

Image# 44 - The Definition of an "Event"

Image: Etzion & Niblett (2011, p.4)

Role of Multitasking (Time Multiplexing)

Multi-tasking plays a vital role in an event-driven architecture, acting as an ***event-producer*** and as an ***event-consumer***. I discuss this in detail in ***Multi-tasking (Time Multiplexing)***

Multi-tasking also has a number of structural advantages when creating programs

Martin (2009, p. 178) puts it very well, saying it *"... is a decoupling strategy. It helps us decouple **what** gets done from **when** it gets done.... Decoupling **what** from **when** can dramatically improve both the throughput and structures of an application. From a structural point of view **the application looks like many little collaborating computers** rather than one big main loop . This can make the system easier to understand and offers some powerful ways to separate concerns..."*

---

[45] This modeling requires a number of different abstractions of the real world....

[46] *"Separation of Concerns" = "Division of Responsibility"*

[47] Such as via Ethernet Packets and then out into the Internet, maybe to other similar systems or devices... This is the essence of "The Internet of Things" See:

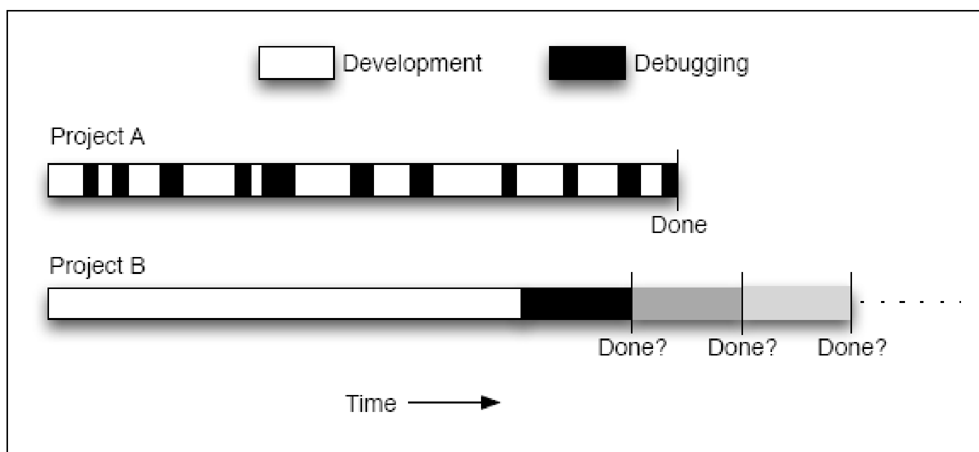
Appendix G. DEATH TO BUGS !!

A Zero-Tolerance Approach to Bugs
A technique I've always found to be very effective is that of ***development-driven testing*** or even ***test-driven development*** where no bugs are tolerated; when one is found it is fixed immediately as part of the development effort

This is in contrast to my many professional experiences in large software projects where most often a developer never tests their own code but another engineer tests it, sometimes weeks later

No Broken Windows!
In the classic book ***The Pragmatic Programmer*** (Hunt & Thomas, 1999), the authors discuss the broken window theory[48] and its relation to a concept of software entropy where small errors left unfixed breed additional errors....

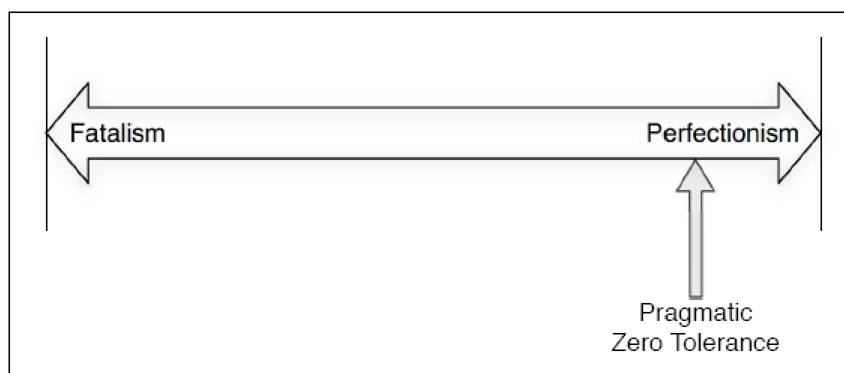
There are many advantages to finding and fixing bugs early apart from the certainty that Butcher highlights (see Image# 45 below); there is the knowledge that very few bugs, if any, exist if a zero tolerance policy to bugs is followed – the "***No Broken Windows***" mindset as Butcher puts it



Image# 45 - Detecting & Fixing Bugs Early Provides Certainty

Image: Butcher (2009, p. 109)

Bugs Prohibited! – Pragmatic Zero Tolerance
Experience teaches us to avoid perfectionism as it is an impossible goal for a human to achieve, so a ***zero tolerance policy towards bugs should be perfectionism tempered by pragmatism***

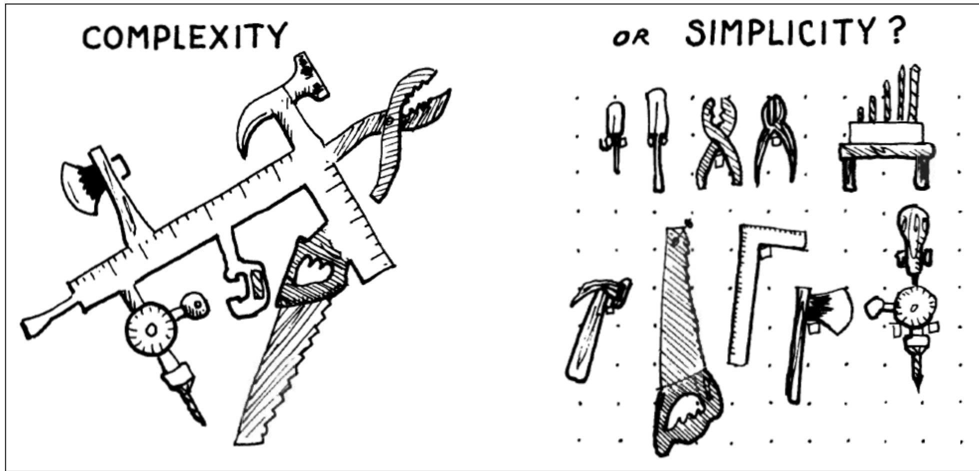


Image# 46 - A Mindset for Debugging...

Image: Butcher (2009, p. 113)

48 James Q. Wilson and George L. Kelling. "Broken Windows: The Police and Neighbourhood Safety," Atlantic Monthly, 1992
http://en.wikipedia.org/wiki/Broken_window_theory

Appendix H. TESTING & DEBUGGING



Image# 47 - Choose your tools carefully – When necessary create your own tools...

Image: (Brodie, 1981, p. 312)

I had intended to put quite a bit of testing & debugging detail here, but ended up doing the exact reverse !

See: ***Method#1*** ***Method#2*** ***Method#3 Implicit Test Scaffolding & Unit Testing*** & ***Method#4***

See also: ***Death to Bugs !!*** & ***Static Analysis using "SPELL"***

I am of the view that it is a ***BIG MISTAKE*** to allow your development project to be ***determined and/or driven by the tools you have available*** !!

If the available tools are inadequate, ***MAKE YOUR OWN TOOLS*** !!
Any competent engineer should be able to make their own tools !!

This what I have had to do as detailed in this document, and it is not as daunting as it seems...

Appendix I. Static Analysis using "*SPELL*"

The Forth Decompiler called "***Spell***" I have written is a very valuable tool to display and analyse the code produced by Forth. Why would you want to do that? Well, part of Forth programming is creating new compilers, and a new compiler will create new structures of some kind in memory, and you need to see if that is happening correctly...

For example, when I made a "black box" that controlled a scanning radio receiver, I had to write some words to create memory arrays and scan arrays and to do that I had to write some memory band and scan band compilers...

In the example below, and purely as an example, I will use the ***Colon Compiler***[49] to define a new word ("*colon definition*") called ***SQUARE*** and analyse its internal form by using the **SPELL** decompiler tool

```
: SQUARE DUP * ; ok

SPELL SQUARE    Ver 0.12

 NFA E300  86      Length-byte
     E301  53       S
     E302  51       Q
     E303  55       U
     E304  41       A
     E305  52       R
     E306  C5       E
 LFA E307  C6 25   Link to VLIST
 CFA E309  91 0E   (:)
 PFA E30B  95 0C   DUP
     E30D  C1 19   *
     E30F  0B 0B   ;S
```

Indirectly Threaded Code consisting of ***a list of execution addresses*** (***code field addresses*** ("***CFA***'s")

```
0E81 C1                 DB   0C1H
0E82 BA                 DB   ':'+80H
0E83 6A0E               DW   EXITOP-8
0E85 910E      COLON:   DW   DOCOL
0E87 D412               DW   QEXEC      ; ?EXEC
0E89 8F12               DW   SCSP       ; !CSP
0E8B 7917               DW   CREAT      ; CREATE
0E8D 4013               DW   RBRAC      ; ]
0E8F A413               DW   PSCOD      ; (;CODE)
0E91 2AFBFF    DOCOL:   LD   HL,(RPP)
0E94 2B                 DEC  HL
0E95 70                 LD   (HL),B
0E96 2B                 DEC  HL
0E97 71                 LD   (HL),C
0E98 22FBFF             LD   (RPP),HL
0E9B 13                 INC  DE
0E9C 4B                 LD   C,E
0E9D 42                 LD   B,D
0E9E C3 53 07           JP   NEXT
```

(i)

```
SPELL DUP     Ver 0.12

 NFA 0C8F  83      Length-byte
     0C90  44       D
     0C91  55       U
     0C92  D0       P
 LFA 0C93  79 0C   Link to 2SWAP
 CFA 0C95  97 0C    Machine Code
 PFA 0C97  E1 E5
   Etc...
```

(ii)

```
SPELL *     Ver 0.12

 NFA 19BD  81      Length-byte
     19BE  AA       *
 LFA 19BF  96 19   Link to M/
 CFA 19C1  91 0E   (:)
 PFA 19C3  80 19   M*
     19C5  56 0C   DROP
     19C7  0B 0B   ;S
```

(iii)

```
0B06 82                 DB   82H
0B07 3B                 DB   ";"
0B08 D3                 DB   'S'+80H
0B09 ED0A               DW   RPSTO-6
0B0B 0D0B      SEMIS:   DW   $+2
0B0D 2AFBFF             LD   HL,(RPP)
0B10 4E                 LD   C,(HL)
0B11 23                 INC  HL
0B12 46                 LD   B,(HL)
0B13 23                 INC  HL
0B14 22FBFF             LD   (RPP),HL
0B17 C3 53 07           JP   NEXT
```

(iv)

Image# 48 - Using the Colon Compiler to create a simple definition & analysing the threaded code produced

Image: Husband, 2011, using keyboard activity, "SPELL" Forth Decompiler Tool & eZ80 code based on the Intel 8080 figForth model

The word : ("***colon***") in (i) is a ***colon definition***, as is ***** ("***star***") in (iii)

The words ; ("***semi***") in (iv) and ***DUP*** in (ii) are ***code definitions*** ("***primitives***")

SQUARE is a very simple high-level colon definition, which takes a stack item, squares it (multiplies it by itself) and replaces the result onto the stack

It does this by making a copy of the single stack item with ***DUP*** and using ***** ("star") to multiply them together, leaving the result back on the stack

[49] Forth contains a number of compilers (and interpreters) and new compilers and interpreters can be added as required...

Decompilation of the word ***SQUARE*** by ***SPELL*** exposes its *threaded code* internal form....

(1) ***NFA*** marks the start of the ***Name Field Address*** of SQUARE's header

(2) ***LFA*** marks the ***Link Field Address*** used for threading the dictionary's linked list

(3) ***CFA*** marks the ***Code Field Address*** which always points towards *executable code* (*machine code*)

(4) ***PFA*** marks the ***Parameter Field Address*** which is the start of a field of data which is interpreted by the code pointed to by the ***CFA***...

In the case of a high-level ***colon definition***, that data is a list of the ***Code Field Addresses*** of the other words in the high-level definition...

In the case of a *primitive definition* (***code definition***) the ***CFA*** points towards *its own* ***PFA*** (thereby complying with "rule" (3) above; the ***PFA*** contains executable machine code

In the case of other kinds of definitions, such as **constants**, or **variables**, the ***PFA*** will contain data that will be interpreted appropriately....

Appendix J. Configuration Management - Baselines & Versioning

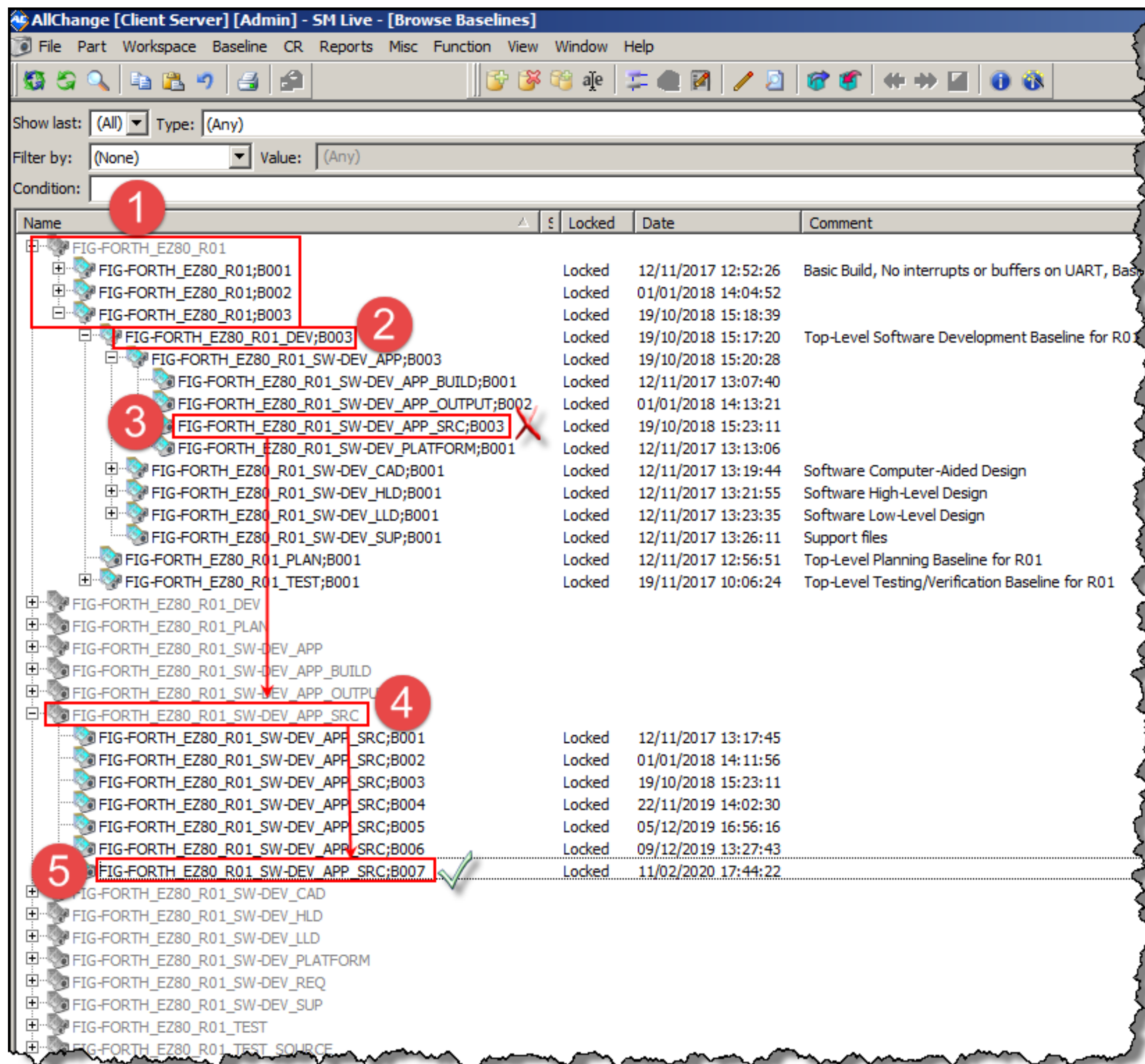
In a previous life, when I sold multi-tasking Forth up-grades for home computers and then went on to sell "black-boxes" that decoded data-over-radio, etc, I wrote a lot of software after designing the hardware...
(See: ***Past Products & Activities***)

I always struggled with a particular aspect of the design/software engineering but I never really knew what it was[50] and what form the solution might take...

Many years later, when I worked at Agusta-Westland in Yeovil, I was introduced to AllChange, ("***A/C***") a configuration management tool/database[51] and much more[52]. It was "love-at-1st-sight"[53] and I have my own personal licence... I am using A/C "straight-out-of-the-box" without using any scripting...

Meta Baselines

A "meta" baseline is a baseline[54] that contains other baselines...
So, in this case, my ***master meta-baseline*** is "**FIG-FORTH_EZ80_R01**"   See Point 1 in Image# 49 below
The name of the master meta-baseline is "greyed-out" because it is a "virtual baseline" that can/will contain a number of ***versions*** of itself... Why? Because all baselines change and this is a "change management" tool...



Image# 49 - The contents of the FIG-FORTH_EZ80_R01 Master ("Meta") Baseline

[50] I now realise that I was lacking the ability to perform and manage ***Configurations***, ***Baselining***, ***Versioning*** and implicitly, being able to perform ***Reversion***, and creating and preserving ***Design Archives***...

[51] **Ethical Note**: I am a happy, satisfied customer, who pays Intasoft in Exeter for an annual personal A/C License...

[52] It is highly scriptable and the person @ Westlands who was responsible for it was "underemployed" and was happy to write scripts to implement "team management" and other "business" functionality... So I described the functionality and he wrote the scripts!

[53] And we all know that feeling... !!

[54] "*a baseline is an agreed description of the attributes of a product, at a point in time, which serves as a basis for defining change. A change is a movement from this baseline state to a next state. The identification of significant changes from the baseline state is the central purpose of baseline identification...*"
https://en.wikipedia.org/wiki/Baseline_(configuration_management)

Having a "***version***" is a means of controlling and documenting change[55] within a baseline...

At the moment (while in the development stage) the baseline I am interested in is "**FIG-FORTH_EZ80_R01_DEV**" as shown in Point 2 in Image# 49 above

This is the "***DEV***" metabaseline which contains a further 5 meta-baselines. I am interested in the first of these five; "**FIG-FORTH_EZ80_R01_SW-DEV_APP**" as shown underneath Point 2 in Image# 49 above and this meta-baseline contains another 4 baselines...

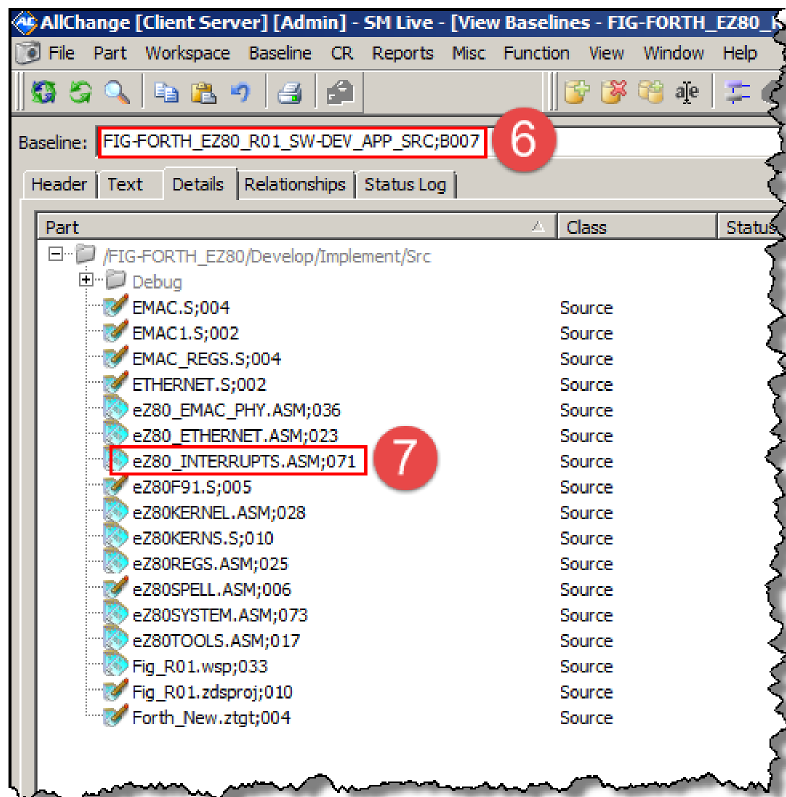
Out of those 4 baselines, I am working in "***FIG-FORTH_EZ80_R01_SW-DEV_APP_SRC***" as shown in Point 3[56] in Image# 49 above and it can be seen at Point 4 in Image# 49 above

In Point 4, it can also be seen that this baseline contains 7 ***locked***[57] versions... V7 being the state of work as of the 11th February 2020... as seen in Point 5

What this means is that I am currently working on ***parts*** that have not yet been baselined[58] into a "***B008***" version...

B007 ("Build 007") Baseline Contents

So, I am taking about baseline "***FIG-FORTH_EZ80_R01_SW-DEV_APP_SRC;B007***" as seen in Point 6 of Image# 50 below. And this IS NOT a meta-baseline as it contains ***REAL PARTS*** !!



Image# 50 - The contents of the Build 7 eZ80 Source Baseline

So, up to now, I've been using A/C in its ***browsing*** and ***viewing baseline*** modes. Turning to Image# 51, we now see a parts view[59] of the A/C workspace...

The ***A/C Workspace*** I am concerned with is shown in Image# 51, Point 10 below... This is mapped onto my "P-Drive" on my server[60]. A/C refers to the folders below the top-level as "sub-systems" as in Point 11, and the selected sub-system is shown in Point 9 and displayed on the right...

---

[55] A component of software configuration management, ***version control***, also known as **revision control** or **source control**,[1] is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". ***https://en.wikipedia.org/wiki/Version_control***..... A/C calls it a "***Version***". Its naming format is determined by the user...

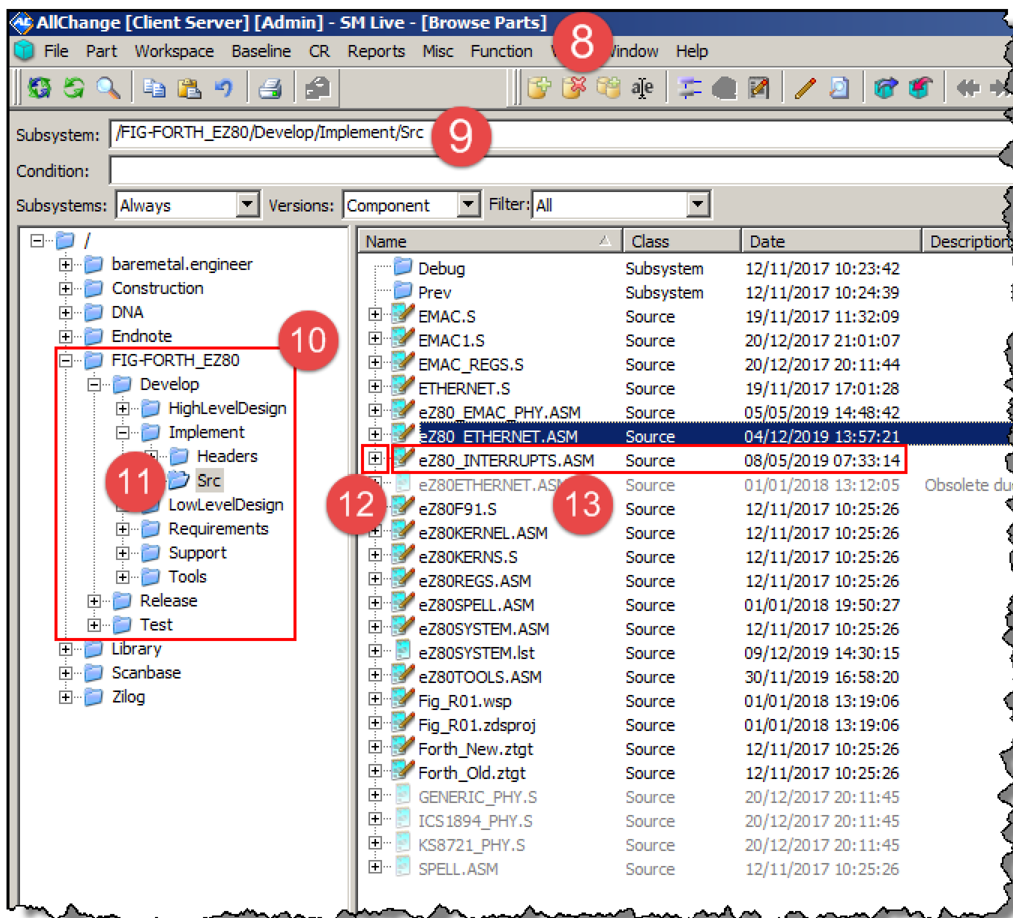
[56] Spot the omission! Baseline ***FIG-FORTH_EZ80_R01_SW-DEV_APP*** should contain ***B007*** of baseline ***FIG-FORTH_EZ80_R01_SW-DEV_APP_SRC*** *if* it were up-to-date...

[57] When a baseline is "***locked***" it is closed and cannot be altered...

[58] This is not a problem for me, but in a commercial environment, the current baseline would be defined 1st as part of the scoping of the work for that iteration... A/C can be set-up to automagically populate and update the current baseline as team members check files out and then check them in again...

[59] These different views are selected from the toolbar offering "***File***" "***Part***" "***Workspace***" "***Baseline***", etc. choices as shown in Image# 51, Point 8

[60] If I wanted to view that file structure and those files directly, I could use the "***File***" option... A/C is happy to work with any existing file/folder structures you may already have. You can also have multiple workspaces and choose between them...



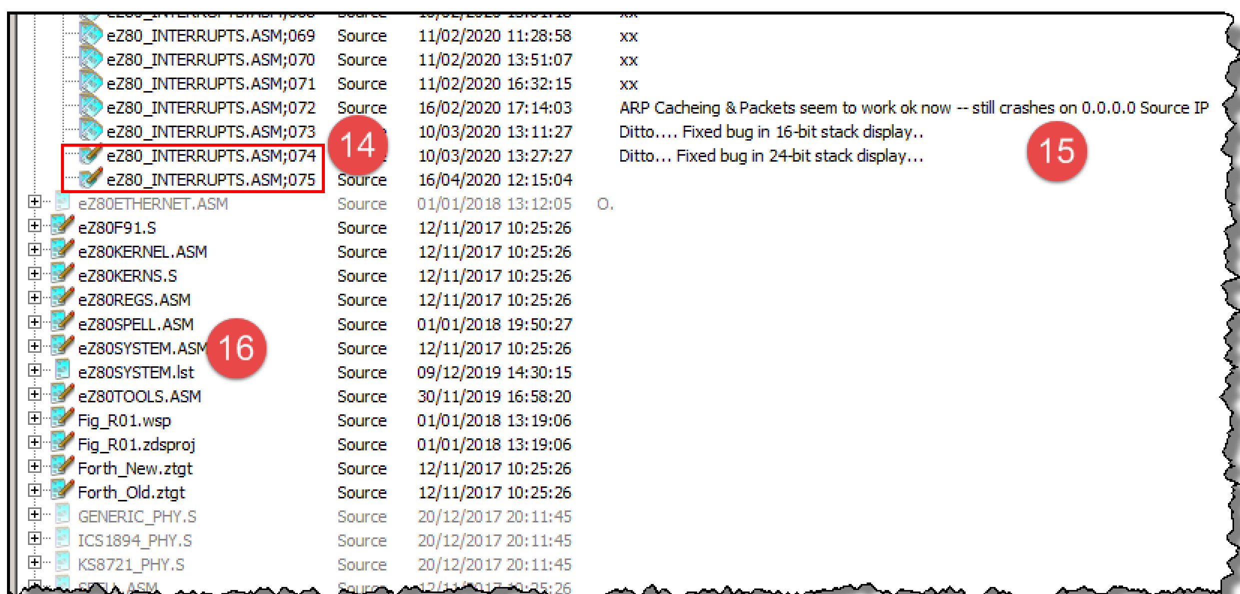
Image# 51 - A "Parts" view of current source development work...

Most of my work is currently being carried out in ***eZ80_INTERRUPTS.ASM***, as shown in Point 13. The system needs all the other files, so they are all checked-out, too (Point 16)

Versioning[61]

If I click on the "+" against the selected file as shown in Point 12, the display will open-up for that file and show all its versions[62] See Image# 52 below

In this way, A/C keeps track of all the changes going on. The user chooses when to check-out and to check back in again...



Image# 52 - A view of Parts Versioning...

---

[61] An important "side-effect" of versioning is that it provides visual proof of a development process as a counter to possible future patent or plagiarism claims...

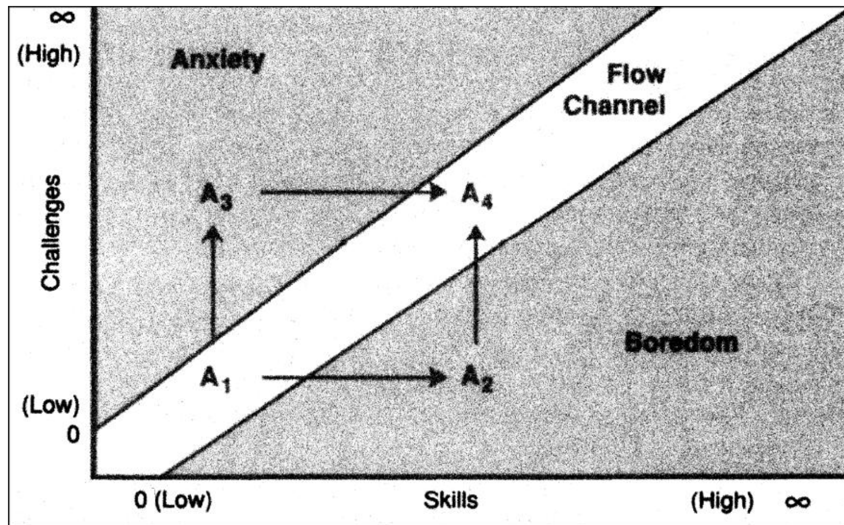
[62] These versions are created automagically as a result of the "checking-out, making changes, checking-in" sequences

Appendix K. "IN THE FLOW"

I am a VERY experienced, self-taught hardware & hardcore software engineer with an "Autotelic" personality...
When programming/testing I always aim to be "***in the flow***" ...

"Mihaly Csikszentmihalyi describes people who are internally driven, and who as such may exhibit a sense of purpose and curiosity, as autotelic. This is different from being externally driven, in which case things such as comfort, money, power, or fame are the motivating force. Csikszentmihalyi writes:

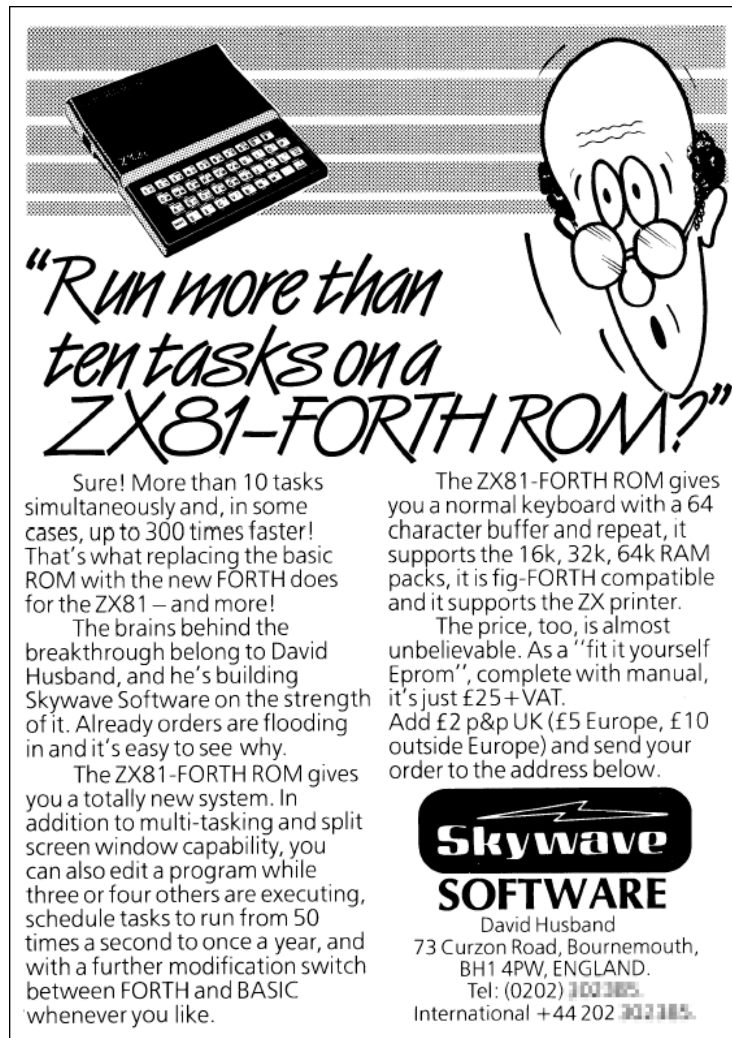
*"An autotelic person needs few material possessions and little entertainment, comfort, power, or fame because so much of what he or she does is already rewarding. Because such persons **experience flow** in work, in family life, when interacting with people, when eating, even when alone with nothing to do, they depend less on external rewards that keep others motivated to go on with a life of routines. They are more autonomous and independent because they cannot be as easily manipulated with threats or rewards from the outside. At the same time, they are more involved with everything around them because they are fully immersed in the current of life""*



Why the complexity of consciousness increases as a result of flow experiences

Image# 53 - The Flow Channel

Appendix L. PAST PRODUCTS & ACTIVITIES



Image# 54 - Advert for the Multitasking ZX-81 Forth ROM

Image: (Husband, 1984)

1984 - Multitasking ZX81 Forth

The revolutionary ZX81 computer by Clive Sinclair came out in 1981 and quickly sold in large quantities. Sinclair claimed (rather tongue-in-cheek) that you could use it to control a power station. It was based on a Z80 running at about 2 Mhz. It had an 8k basic

I sold a replacement for the internal Basic ROM that converted the ZX81 into a multitasking Forth machine. It could even run overlapping multitasked windows which was rather revolutionary at the time because it was some time before Microsoft came out with Windows 3.0 which did the same thing on a PC

1985 - Multitasking BBC Forth-83

What the competition hasn't been waiting for.

Latest version of Forth for the BBC (Is not rehashed Forth 79 Code)

Unique Stack Display Utility

16k Eprom type 27128

Multi-tasking operating system for Real-Time use.

Here's the Forth Eprom for the BBC Micro that makes all others out of date.

It's Multi-Forth 83 from David Husband who has built his reputation for Quality Forth products with his ZX81-Forth ROM, Spectrum Forth-I/O Cartridge and now New Multi-Forth 83 for the BBC Micro. This is not rehashed Forth 79 Code, but a completely new version of the Forth 83 Standard. It's unique in that it Multi-tasks, and therefore the user can have a number of Forth programs executing simultaneously and transparently of each other.

Multi-Forth 83 sits in the sideways ROM area of the BBC along with any other ROMs in use. It is compatible with the MOS, and specially vectored to enable a system to be reconfigured. It contains a Standard 6502 Assembler, a Standard Screen Editor, and a Unique Stack Display Utility.

With this Forth, David Husband has provided the BBC Micro with capabilities never before realised. And being 16K rather than 8K is twice the size of other versions. Multi-Forth 83 is supplied with an extensive Manual (170 pages plus) and at £45+VAT it is superb value.

Order it using the coupon adding £2.30 p&p (£5 for Europe, £10 outside) or if you want more information, tick that box instead. Either way, it will put you one step ahead of the competition.

Please send me Multi-Forth 83 for BBC Micro. £45+VAT. De-luxe System inc. Disc £80+VAT. Cheques to Skywave Software Readers' A/C (or enter Visa No.)

Please send me more information:
- Multi-Forth 83
- ZX81-Forth ROM
- Spectrum Forth-I/O Cartridge

Name__________

Address__________

Post code__________

SUBJECT TO AVAILABILITY. FOR I.O.O.S ONWARDS.
Send to Skywave Software, 73 Curzon Road, Bournemouth, BH1 4PW, Dorset, England. Tel: (0202)

Skywave SOFTWARE

MULTI-FORTH 83 FOR THE BBC MICRO

Image# 55 - Advert for the Multitasking BBC Forth 83 ROM

Image: (Husband, 1985)

This Forth was sold as a plug-in eprom for the BBC computer and as a ROM Cartridge for the Acorn Electron

A NEW TOOL FOR A NEW TECHNOLOGY

Scanmaster

Spectrum Planning? Surveillance? Management? Activity Monitoring? System Engineering?

Why not upgrade your Scanning Receiver with the New SCANMASTER™ II Controller?

Scanmaster™ II is a "black box" which plugs into the Remote Socket on your Scanning Receiver and takes over control of the Radio. No programming is required to use **SM2™** as it has its own powerful commands to perform Searching, Memories, Remarks, Logging, Activity Reports, and many other things. It generates **INFORMATION** and you can control what sort and how. This information can be uploaded, downloaded, printed, stored, etc. You talk to **SM2™** with a Terminal via an **RS232** Serial Port. This would usually be **ANY** Computer running a simple terminal/comms program. Anything to or from the terminal is automatically stored in spare memory, backed with a lithium battery, and can be viewed at any time, even if **SM2™** has been turned off for a while. **SM2™** uses the latest microprocessor technology and is designed to be **User Friendly**. **SM2™** has its own Clock/Calendar with a lithium battery so it keeps the date/time even when turned off. **SM2™** has a powerful **Scheduler** so that jobs can be set up to run on a certain date/time, at date/time intervals, for a date/time duration, or for a number of times. **SM2™** runs from 12v D.C. and is ideal for vehicle or portable use. It is **LOW-COST** so every Radio Engineer should have one!

SM2™ is designed to be **upgradable**, and will run many different radios with just a change of software and connecting leads. It will accept **Plug-In Boards** to further expand its power and usefulness. Once you have used **SM2™** you will wonder how you ever used a Scanning Receiver without it !!

- Powerful Activity Reports
- Over 1300 Memories
- Clock/Calendar (Lithium)
- Versatile Lockouts
- Output for Databases (Comma Delimited)
- Powerful Scheduler
- Assign Remarks
- Low cost, easy to use
- Sophisticated Logging
- User Friendly Commands
- Large Memory + Battery
- Remote/Unattended Use
- Parallel Printer Port
- Status Indicator Panel
- Many Search Bands
- Tape Recorder On/Off
- Extensive User Manual
- Takes Plug-In Boards
- P.C. Upload/Download

Many more features !! Ask for more information.

SM2™ is supplied complete with connecting leads and a software package fitted to drive one the following Scanning Receivers:

AR3000 AR2002 CHASE IC-R7000 IC-R9000 FRG9600
(Upgrades are available from time to time)

How to Order: (Specify type of Receiver & Computer)
Send Cheque, Bank Draft, or Mastercard/Access/Visa No. & Exp. Date. Official Orders accepted from Govn't, Education, Large Co.s, etc.

Special Price....£249.99

Delivery: Ex Stock, subject to availability

28 Day Money-Back Guarantee if not entirely satisfied.
Life-Time Service Guarantee (Subject to conditions)

EMP
L I M I T E D
Street, Portland, Dorset
DT5 1JQ, England
Phone: (0305)

Overseas Agents Wanted

Scanmaster™ II Plug-in Boards

SM2™ is able to take extra internal plug-in boards & software packages to further expand its power and usefulness. All boards except **ROMCARD™** contain their own operating software and will work with any of the main **SM2™** software packages thereby allowing the various boards to work with any Scanning Receiver supported by us.

Available Now...

ROMCARD™ enables **SM2™** to hold up to 8 different software packages so that **SM2+ROMCARD** can drive up to **8** different Scanning Receivers with only a change of connecting leads. Upon performing a cold start, **SM2™** is able to detect if **ROMCARD™** is fitted and if so, it displays all the software packages fitted on **ROMCARD™** on a menu for user selection. Supplied with one software package and set of connecting leads for **£49.99**
Software available for:

AR3000 AR2002 CHASE IC-R7000 IC-R9000 FRG9600

Available Soon!!

TONECARD™ is able to **Decode** & **Encode** a variety of signalling tones including **DTMF**, **SELCALL**, **CTCSS** & **FFSK**. (MPT1317, etc) and also decode and action signalling protocols such as **TRUNKING** (MPT1327, etc), **JRC Band**, **BANDIII**, etc. **SM2+TONECARD** can drive a transceiver if you want. **SM2+TONECARD** can control more than just Scanning Receivers!

JRC ROM™ is an optional software package for **TONECARD** and is tailor made to support the new **JRC Band** signals used by the Gas & Elect. Co.s. Many powerful features for diagnostic and engineering uses.

CELLCARD™ contains the same modem/filter chipset as a cellular telephone and **SM2+CELLCARD™** can decode the signalling protocols used on the UK **TACS** system and optionally, the **AMPS** system. It can decode data and commands and can do channel hopping and other things. **CELLCARD™** can be used as a powerful diagnostic tool and is available to **AUTHORISED USERS ONLY !!**

SCANMASTER™ I ??
The Old Original, still going strong and still available for the
AR2002, IC-R7000 & FRG9600
for only **£153.26**

Image# 56 - Advert for the Scanmaster II Scanner Controller

Image: (Husband, 1990)

Tools for Radio Engineers

Only
£699
Plus VAT

SELCALL . CTCSS . DTMF . FFSK . MPT1327

A New Tool for a New Technology

Scanmaster™ **decodes** & encodes **SELCALL, CTCSS, DTMF, FFSK & TRUNKING**. **Scanmaster™** will drive and control most popular scanning receivers including **R7000, FRG9600 AOR3000/A, AOR2002** and **CHASE**. Engineers can communicate with **Scanmaster™** using any computer or Laptop, running a standard **terminal** program via an **RS232** Port. **Scanmaster™** operates on a **12v DC** supply making it ideal for use in all workshop, vehicle or field locations.

Scanmaster™ is a powerful, yet **User Friendly** low-cost diagnostic tool that will enable engineers to select, **monitor** and generate **system/traffic** activity and load information using the inbuilt commands that perform functions including -:

Searching - Memories - Logging
Remarks - Activity Reports

Scanmaster™ incorporates a lithium battery that ensures all information and schedules are **automatically stored** in its own powerful memory with Date/Time logging to assist in detailed analysis.

Scanmaster™ will support the new **JRC** and **Water Industry Trunking** Schemes as standard, including **Scottish Power** (Bona-fide users only) **Scanmaster™** has many more features; far too many to list here and carries a **Lifetime Service Guarantee**.
Delivery:- **Ex-stock** subject to availability.

ORDER NOW!!

Call or fax for more information.
Versions for **CELLULAR** & **PAGING** coming soon!!

Scanmaster Street, Portland, Dorset. DT5 1 JQ Tel 0305 Fax 0305

Image# 57 - Advert for the Scanmaster II Professional Communications Decoder

Image: (Husband, 1991)

Mobile monitoring tool

Trunking English™ again, this time a BCAST Sysdef 5, which is inviting radio units to sample signals on adjacent control channels.

This is quite a useful message because it tells us the channel numbers of adjacent control channels and the sys id of the adjacent nodes.

This is the system identity code of the site sending the whole sequence of messages ("a frame").

These are the System Identity Codes of the adjacent nodes on the channel numbers mentioned.

```
ALH Invitation to 0/0_DummyI via 14578
BCAST 5 Vote Now on Ch 89 via 14584
ALH Invitation to 0/0_DummyI via 14578
BCAST 5 Vote Now on Ch 93 via 14472
ALH Invitation to 0/0_DummyI via 14578
BCAST 5 Vote Now on Ch 89 via 14584
ALH Invitation to 0/0_DummyI via 14578
ALH Invitation to 0/0_DummyI via 14578
```

This annotated extract from a Scanmaster display illustrates signalling on a time-shared control channel in a public utility radio system. The unit has been set to hide all information other than the idling frames

A POWERFUL diagnostic tool for engineers and technicians who install or maintain complex radiocommunications systems has been developed by the Dorset firm Scanmaster Products.

The microprocessor-controlled Scanmaster unit connects between a scanner receiver (four common types are supported) and a laptop computer terminal, to provide intelligent control of the receiver. The whole system can be battery powered, for use on site as well as in the workshop.

At its simplest, the Scanmaster unit enables the operator to select and monitor a radio channel by typing in the channel number: there is no need to remember the exact frequency because the Scanmaster can calculate it from a programmed-in translation table. Keyboard short-cuts can be used for stepping channels up or down.

But the system can go much further than this, using plug-in cards to decode information from the receiver's audio output and act upon it. The "Tonecard", for example, decodes MPT 1327 data messages on trunked radio channels, revealing precisely what is happening on the network. When a call is set up, the user can make the receiver switch from control channel to traffic channel and back, manually or automatically, so that the progress of the call can be tracked in detail.

Included in the Tonecard are decoders for CTCSS, Selcal, DTMF and DCS. Plug-in roms can be added to adapt the card for proprietary systems or specific radio schemes — examples are the water industry's and the fuel and power industry's. Other cards have been developed for cellular and paging applications; they stack together so that several are available at once.

A feature of the screen display is what David Husband, the system's designer, calls Trunking English. In place of the raw hexadecimal codes of the MPT 1327 messages are plain language phrases. By reading them as they scroll up the screen, the user can read and understand the signalling in real time.

For diagnosing problems, the card can be reconfigured in moments so that it displays only the message groups of interest. The terminal software can capture this output in a computer file for further study.

"The engineers haven't got anything like this, and they need it desperately", said Mr Husband, who claims that the £700 Scanmaster has capabilities not available even on £20 000 test sets. He adds that it is also valuable as a training aid. Scanmaster is available direct from the manufacturer to bona fide users only.

Image# 58 - Review of MPT1327 Decoding by the Scanmaster II

Image: (Husband, 1993, p. 40)