The HCCS Forth ROM for the Epson HX-20

Martin Hepperle, May 5th, 2022

These are my notes on recreating the assembler code of the Forth ROM as well as the systems Basic ROMs. They worked for me, but may be incorrect – use them at your own risk.

In the early 1980s, HCCS Associates in Newcastle upon Tyne, England (1983-1996) sold a Forth language interpreter for the Epson HX-20 handheld computer. This Forth implementation was written by Joseph William Brown (also the author of [1]) and distributed as an EPROM to be plugged into the HX-20 or its RAM/ROM expansion box. It was directly based on figForth for the 6800 processor [3]. However, it was extended with specific words, supporting specific features of the HX-20. Also, some Forth-79 words, like `U.`, can be found. The specific words provided access to internal and external cassette tapes as well as commands to draw points and lines on the LCD screen and, of course, an important BEEP word. Furthermore, support for the built-in printer and date and time derived from the HX-20 clock were added. Also included was a simple screen editor.

The inheritance from figForth is important to remember, as this affects some of the inner structures and algorithms, e.g. when searching vocabularies. Many books of that era include sections about the differences between figForth, Forth-79 and Forth-83.

The same software developer created a Forth implementation for the B.B.C. Microcomputer, which was very popular in the UK at the time. Additional implementations followed. The books [1] and [2] can be used as more detailed references – many of the Forth words in these systems are identical. Additional information on Forth can be found in many texts ([3] still provides an excellent introduction into Forth) which often distinguish between figForth and other variants of Forth.

Bibliography

[1] J. W. Brown, "Forth for the B.B.C. Microcomputer", HCCS Associates, 533 Durham Rd., Low Fell, Gatesland, England, 1982.

[2] R. deGrandis-Harrison, "Forth on the B.B.C. Microcomputer and Acorn Electron", Acornsoft Limited, 4a Market Hill, Cambridge, CB2 3NJ, England, 1983.

[3] fig-FORTH FOR 6800, Assembly Source Listing, Release 1, Forth Interest Group, San Jose, USA, May 1979.

[4] L. Brodie, “Starting Forth”, Prentice Hall Inc., Englewood Cliffs, New Jersey 07632, 2nd edition, 1987.

[5] Epson HX-20 - Technical Manual – Software.

[6] http://electrickery.xs4all.nl/comp/hx20/

Different ROM Versions

Instead of relying solely on the official entry points, the Forth ROM makes use of some undocumented system routines embedded in the two Basic ROMs. These ROMs exist in different versions. The Forth Rom was written for the `JAPAN V1.0` ROMs, requiring adaptation of the Forth ROM if your HX-20 has different system ROMs.

The first Basic ROM, mapped into the memory range C000-DFFF, appears in two versions `V1.0` and `V1.1`. These versions differ, but here is no difference between the Japanese/US/UK systems and the European systems. The second BASIC ROM, covering the range E000-FFFF, also exists in both versions `V1.0` and `V1.1` but these also differ for the Japanese/US/UK and the European systems. Therefore you should first use the `MONITOR` first to check the signature of some routines to find out, which Basic ROMs you have. Use the `D` command to look at one or two ROM locations in the ROM address ranges listed in Table 5. Then select the appropriate patch data. Otherwise you may fall into a Trap!

ROM - Address	chk-8	crc-32	crc-16	Usage	Comments
0x8000_9FFF	54	33FBB1AB	336A	BASIC 2	same as Europe V1.0
0xA000_BFFF	5F	27D743ED	E1C5	BASIC 1	same as Europe V1.0
0xC000_DFFF	10	F5CC8868	A310	Menu, Monitor	same as Europe V1.0
0xE000_FFFF	F2	ED7482C6	E7F2	I/O	unique

Table 1: Japan ROMs V1.0.

ROM - Address	chk-8	crc-32	crc-16	Usage	Comments
0x8000_9FFF	54	33FBB1AB	336A	BASIC 2	same as Japan V1.0
0xA000_BFFF	5F	27D743ED	E1C5	BASIC 1	same as Japan V1.0
0xC000_DFFF	10	F5CC8868	A310	Menu, Monitor	same as Japan V1.0
0xE000_FFFF	20	A35DB3FC	A0B1	I/O	unique

Table 2: Europe ROMs V1.0.

ROM - Address	chk-8	crc-32	crc-16	Osage	Comments
0x8000_9FFF	C3	4DE0B4B6	E239	BASIC 2	same as Europe V1.1
0xA000_BFFF	22	10D6AE76	76E2	BASIC 1	same as Europe V1.1
0xC000_DFFF	7D	26C203A1	F3E9	Menu, Monitor	same as Europe V1.1
0xE000_FFFF	7E	101CB3E8	8ED7	I/O	unique

Table 3: Japan ROMs V1.1.

ROM - Address	chk-8	crc-32	crc-16	Usage	Comments
0x8000_9FFF	C3	4DE0B4B6	E239	BASIC 2	same as Japan V1.1
0xA000_BFFF	22	10D6AE76	76E2	BASIC 1	same as Japan V1.1
0xC000_DFFF	7D	26C203A1	F3E9	Menu, Monitor	same as Japan V1.1
0xE000_FFFF	F4	FD339AA5	2F07	I/O	unique

Table 4: Europe ROMs V1.1.

Basic ROM	Bytes at Address	V 1.0 Japan	V 1.0 Europe	V 1.1 Japan	V 1.1 Europe
Basic ROM C000-DFFF	D760	53 D5		43 D4	
Basic ROM E000-FFFF	E2EF	3C 37	18 25	E2 3F	81 20

Table 5: These byte sequences characterize the two Basic ROMs. The shaded cells highlight the reference version – the unmodified Forth ROM can be used only with these ROMs.

If you are more comfortable with Basic, you can use the following program to detect the version of your ROMs:

```
10 C$="UNKNOWN"
20 D$="UNKNOWN"
30 B=PEEK(&HD760)
40 IF B=&H53 C$="JAPAN/EUROPE 1.0"
50 IF B=&H43 C$="JAPAN/EUROPE 1.1"
60 B=PEEK(&HE2EF)
70 IF B=&H3C D$="JAPAN 1.0"
80 IF B=&HE2 D$="JAPAN 1.1"
90 IF B=&H18 D$="EUROPE 1.0"
100 IF B=&H81 D$="EUROPE 1.1"
110 PRINT "C000-DFFF=";C$
120 PRINT "E000-FFFF=";D$
130 END
```

Listing 1: Basic program to identify the three ROM versions.

You can create a modified ROM image for your system with any decent HEX editor. The following tables list the addresses and the required modifications for the different ROMs. Or use one of the four preconfigured ROM images provided with these notes.

The Basic ROM at C000-DFFF

Forth address `$6082` points to a message at `D735`.	
JAPAN V1.0, EUROPE V1.0	EUROPE V1.1, JAPAN V1.1 Change to `$D744`.
`D735: 00 01 (row,col coordinates)` `D737: 45 72 72 6F 72 "Error"`	`D744: 00 01 (row,col coordinates)` `D746: 45 72 72 6F 72 "Error"`

Forth addresses `$6087` and `$70F2` call `$D715`.	
JAPAN V1.0:	EUROPE V1.1: Change to `$D724`.
`D715: 7E FF 49 jmp LFF49`	`D724: 7E FF 49 jmp LFF49`

Forth address $74A9 calls $D310.

JAPAN V1.0, EUROPE V1.0:

```
D310: 8D 46    bsr   LD358
D312: 30       tsx
D313: 08       inx
D314: DF 6E    stx   X006E
D316: 38       pulx
D317: 3C       pshx
D318: FF 02 BF stx   X02BF
D31B: FF 02 C1 stx   X02C1  ; RTNADDR
D31E: B6 02 80 ldaa  X0280
D321: B7 02 A3 staa  X02A3
D324: BD D7 14 jsr   LD714
D327: 20 1D    bra   LD346

LD329:
D329: 8D 2D    bsr   LD358
D32B: 30       tsx
D32C: EC 05    ldd   $05,x
D32E: FD 02 BF std   X02BF
D331: C6 06    ldab  #$06
D333: 3A       abx
D334: DF 6E    stx   X006E
D336: BD D7 14 jsr   LD714
D339: FE 02 A0 ldx   X02A0
D33C: 26 0D    bne   LD34B
D33E: CE D7 2A ldx   #$D72A   ; Trap!

LD341:
D341: C6 05    ldab  #$05
D343: BD D7 15 jsr   LD715

LD346:
D346: BD D6 62 jsr   LD662
D349: 20 36    bra   LD381

LD34B:
D34B: FC 02 A2 ldd   X02A2   ; BP value
D34E: FE 02 A0 ldx   X02A0   ; BP address
D351: A7 00    staa  $00,x
D353: CE D7 47 ldx   #$D747  ; "Break"
D356: 20 E9    bra   LD341   ; show
```

EUROPE V1.1:

No Change, completely different, Trap!

```
D310: 38       pulx          <<< data on stack
D311: DF 69    stx   X0069
D313: 38       pulx
D314: DF 6B    stx   X006B
D316: 32       pula
D317: 97 6D    staa  X006D
D319: 38       pulx
D31A: DF 6E    stx   X006E

D31C: 8D 31    bsr   LD34F
D31E: BD D7 23 jsr   LD723
D321: DE 6E    ldx   X006E    ;
D323: 8C D7 7E cpx   #$D77E
D326: 27 0F    beq   LD337
D328: B6 02 A2 ldaa  X02A2    ; BP value
D32B: BC 02 A0 cpx   X02A0    ; BP address
D32E: 26 0F    bne   LD33F    ; -> Trap
D330: A7 00    staa  $00,x
D332: CE D7 56 ldx   #$D756   ; "Break"
D335: 20 0B    bra   LD342    ; show

LD337:
D337: 38       pulx
D338: FF 02 C1 stx   X02C1     ; set RTNADDR
D33B: DF 6E    stx   X006E

LD33D:
D33D: 20 08    bra   LD347

LD33F:
D33F: CE D7 39 ldx   #$D739   ; "Trap!"

LD342:
D342: C6 05    ldab  #$05
D344: BD D7 24 jsr   LD724

LD347:
D347: BF 02 BF sts   X02BF
D34A: BD D6 71 jsr   LD671
D34D: 20 24    bra   LD373
```

Forth address $7C51 calls $D957.

JAPAN V1.0, EUROPE V1.0:

```
LD957:
D957: 8D 4F    bsr   LD9A8
D959: 8A 80    oraa  #$80
D95B: BD FF 55 jsr   LFF55
D95E: 96 52    ldaa  X0052
D960: 48       asla
D961: 48       asla
```

EUROPE V1.1:

Change to $D967:

```
LD967:
D967: 8D 4F    bsr   LD9B8
D969: 8A 80    oraa  #$80
D96B: BD FF 55 jsr   LFF55
D96E: 96 52    ldaa  X0052
D970: 48       asla
D971: 48       asla
```

Forth address $7C6D calls $D977.

JAPAN V1.0, EUROPE V1.0:

```
D977: 8D 2F    bsr   LD9A8
D979: 37       pshb
D97A: 36       psha
D97B: 86 63    ldaa  #$63
D97D: BD FF 55 jsr   LFF55
D980: 32       pula
```

EUROPE V1.1:

Change to $D987.

```
D987: 8D 2F    bsr   LD9B8
D989: 37       pshb
D98A: 36       psha
D98B: 86 63    ldaa  #$63
D98D: BD FF 55 jsr   LFF55
D990: 32       pula
```

Forth address $7C90 calls $D9D6.

JAPAN V1.0, EUROPE V1.0:

```
LD9D6:
D9D6: DE 50    ldx   X0050
D9D8: A6 01    ldaa  $01,x
D9DA: 36       psha
D9DB: A6 05    ldaa  $05,x
D9DD: 36       psha
D9DE: A6 09    ldaa  $09,x
D9E0: A7 05    staa  $05,x
D9E2: BD D9 57 jsr   LD957
D9E5: DE 50    ldx   X0050
D9E7: 32       pula
```

EUROPE V1.1:

Change to $D9E6.

```
LD9E6:
D9E6: DE 50    ldx   X0050
D9E8: A6 01    ldaa  $01,x
D9EA: 36       psha
D9EB: A6 05    ldaa  $05,x
D9ED: 36       psha
D9EE: A6 09    ldaa  $09,x
D9F0: A7 05    staa  $05,x
D9F2: BD D9 67 jsr   LD967
D9F5: DE 50    ldx   X0050
D9F7: 32       pula
```

Forth address $7CB8 calls $DA07.

JAPAN V1.0, EUROPE V1.0:

```
DA07: DE 50    ldx   X0050
DA09: C6 01    ldab  #$01
DA0B: 3A       abx
DA0C: DF 5C    stx   X005C
DA0E: 8D DE    bsr   LD9EE
DA10: DE 50    ldx   X0050
DA12: C6 03    ldab  #$03
DA14: 3A       abx
```

EUROPE V1.1:

Change to $DA17.

```
DA17: DE 50    ldx   X0050
DA19: C6 01    ldab  #$01
DA1B: 3A       abx
DA1C: DF 5C    stx   X005C
DA1E: 8D DE    bsr   LD9FE
DA20: DE 50    ldx   X0050
DA22: C6 03    ldab  #$03
DA24: 3A       abx
```

The Basic ROM at E000-FFFF

Forth address $6112 calls $E2EF.

JAPAN V1.0:

```
LE2EF:
E2EF: 3C    pshx
E2F0: 37    pshb
E2F1: 81 20 cmpa  #$20
E2F3: 24 0A bcc   LE2FF
E2F5: 81 0A cmpa  #$0A
E2F7: 27 0D beq   LE306
E2F9: 81 0D cmpa  #$0D
E2FB: 27 1C beq   LE319
E2FD: 20 29 bra   LE328
```

EUROPE V1.0:

Change to $E2DB.

```
LE2DB:
E2DB: 3C    pshx
E2DC: 37    pshb
E2DD: 81 20 cmpa  #$20
E2DF: 24 0A bcc   LE2EB
E2E1: 81 0A cmpa  #$0A
E2E3: 27 0D beq   LE2F2
E2E5: 81 0D cmpa  #$0D
E2E7: 27 1C beq   LE305
E2E9: 20 29 bra   LE314
```

JAPAN V1.1:

Change to $E2D4.

```
LE2D4:
E2D4: 3C    pshx
E2D5: 37    pshb
E2D6: 81 20 cmpa  #$20
E2D8: 24 0A bcc   LE2E4
E2DA: 81 0A cmpa  #$0A
E2DC: 27 0D beq   LE2EB
E2DE: 81 0D cmpa  #$0D
E2E0: 27 1C beq   LE2FE
E2E2: 20 29 bra   LE30D
```

EUROPE V1.1:

Change to $E2C0.

```
LE2C0:
E2C0: 3C    pshx
E2C1: 37    pshb
E2C2: 81 20 cmpa  #$20
E2C4: 24 0A bcc   LE2D0
E2C6: 81 0A cmpa  #$0A
E2C8: 27 0D beq   LE2D7
E2CA: 81 0D cmpa  #$0D
E2CC: 27 1C beq   LE2EA
E2CE: 20 29 bra   LE2F9
```

Forth addresses $7BFF and $7EA1 call $E3F2.

JAPAN V1.0

```
LE3F2:
E3F2: 37    pshb
E3F3: 36    psha
E3F4: 72 10 7C oim #$107C
E3F7: 86 30 ldaa  #$30
E3F9: 8D 1D bsr   LE418
E3FB: 32    pula
E3FC: 36    psha
E3FD: 8D 0E bsr   LE40D
```

EUROPE V1.0:

Change to $E3DE.

```
LE3DE:
E3DE: 37    pshb
E3DF: 36    psha
E3E0: 72 10 7C oim #$107C
E3E3: 86 30 ldaa  #$30
E3E5: 8D 1D bsr   LE404
E3E7: 32    pula
E3E8: 36    psha
E3E9: 8D 0E bsr   LE3F9
```

JAPAN V1.1:

Change to $E3DA.

```
LE3DA:
E3DA: 37    pshb
E3DB: 36    psha
E3DC: 72 10 7C oim #$107C
E3DF: 86 30 ldaa  #$30
E3E1: 8D 20 bsr   LE403
E3E3: 32    pula
E3E4: 36    psha
E3E5: 8D 11 bsr   LE3F8
```

EUROPE V1.1:

Change to $E3C6.

```
LE3C6:
E3C6: 37    pshb
E3C7: 36    psha
E3C8: 72 10 7C oim #$107C
E3CB: 86 30 ldaa  #$30
E3CD: 8D 20 bsr   LE3EF
E3CF: 32    pula
E3D0: 36    psha
E3D1: 8D 11 bsr   LE3E4
```

Forth address $7C0E calls $E34D.

JAPAN V1.0:

```
LE34D:
E34D: BD E2 CE jsr LE2CE
E350: CE 00 01 ldx #$0001
E353: 3C       pshx
E354: 3C       pshx
E355: 30       tsx
E356: 72 03 7D oim #$037D
```

EUROPE V1.0:

Change to $E339.

```
LE339:
E339: BD E2 BA jsr LE2BA
E33C: CE 00 01 ldx #$0001
E33F: 3C       pshx
E340: 3C       pshx
E341: 30       tsx
E342: 72 03 7D oim #$037D
```

JAPAN V1.1:

Change to $E332.

```
LE332:
E332: BD E2 B3 jsr LE2B3
E335: CE 00 01 ldx #$0001
E338: 3C       pshx
E339: 3C       pshx
E33A: 30       tsx
E33B: 72 03 7D oim #$037D
```

EUROPE V1.1:

Change to $E31E.

```
LE31E:
E31E: BD E2 9F jsr LE29F
E321: CE 00 01 ldx #$0001
E324: 3C       pshx
E325: 3C       pshx
E326: 30       tsx
E327: 72 03 7D oim #$037D
```

Forth address $7C31 calls $E2A5.			
JAPAN V1.0:	EUROPE V1.0: Change to $E291.	JAPAN V1.1: Change to $E2BA.	EUROPE V1.1: Change to $E276
LE2A5: E2A5: 8D 27 bsr LE2CE LE2A7: E2A7: 16 tab E2A8: 86 11 ldaa #$11 E2AA: BD E4 0D jsr LE40D	LE291: E291: 8D 27 bsr LE2BA LE293: E293: 16 tab E294: 86 11 ldaa #$11 E296: BD E3 F9 jsr LE3F9	LE28A: E28A: 8D 27 bsr LE2B3 LE28C: E28C: 16 tab E28D: 86 11 ldaa #$11 E28F: BD E3 F8 jsr LE3F8	LE276: E276: 8D 27 bsr LE29F LE278: E278: 16 tab E279: 86 11 ldaa #$11 E27B: BD E3 E4 jsr LE3E4

Forth address $7CEC calls $E1FF.			
JAPAN V1.0:	EUROPE V1.0: No change required.	JAPAN V1.1: Change to $E1E0.	EUROPE V1.1: Change to $E1E0.
E1FF: 5F clrb LE200: E200: 3C pshx E201: 8D 0F bsr LE212 E203: A6 00 ldaa $00,x E205: 38 pulx E206: 3C pshx E207: 3A abx E208: A7 00 staa $00,x	E1FF: 5F clrb LE200: E200: 3C pshx E201: 8D 0F bsr LE212 E203: A6 00 ldaa $00,x E205: 38 pulx E206: 3C pshx E207: 3A abx E208: A7 00 staa $00,x	E1E0: 5F clrb LE1E1: E1E1: 3C pshx E1E2: 8D 13 bsr LE1F7 E1E4: A6 00 ldaa $00,x E1E6: 38 pulx E1E7: 3C pshx E1E8: 3A abx E1E9: A7 00 staa $00,x	E1E0: 5F clrb LE1E1: E1E1: 3C pshx E1E2: 8D 13 bsr LE1F7 E1E4: A6 00 ldaa $00,x E1E6: 38 pulx E1E7: 3C pshx E1E8: 3A abx E1E9: A7 00 staa $00,x

Forth address $7EB0 calls $EB8F.			
JAPAN V1.0:	EUROPE V1.0: Change to $EB6E.	JAPAN V1.1: Change to $EB6F.	EUROPE V1.1: Change to $EB57.
LEB8F: EB8F: 86 72 ldaa #$72 EB91: 18 xgdx EB92: B3 02 03 subd $0203 EB95: 2A 08 bpl LEB9F EB97: 43 coma EB98: 53 comb EB99: C3 00 01 addd #$001 EB9C: CE 71 00 ldx #$7100	LEB6E: EB6E: 86 72 ldaa #$72 EB70: 18 xgdx EB71: B3 02 03 subd $0203 EB74: 2A 08 bpl LEB7E EB76: 43 coma EB77: 53 comb EB78: C3 00 01 addd #$001 EB7B: CE 71 00 ldx #$7100	LEB6F: EB6F: 86 72 ldaa #$72 EB71: 18 xgdx EB72: B3 02 03 subd X0203 EB75: 2A 08 bpl LEB7F EB77: 43 coma EB78: 53 comb EB79: C3 00 01 addd #$001 EB7C: CE 71 00 ldx #$7100	LEB57: EB57: 86 72 ldaa #$72 EB59: 18 xgdx EB5A: B3 02 03 subd $0203 EB5D: 2A 08 bpl LEB67 EB5F: 43 coma EB60: 53 comb EB61: C3 00 01 addd #$001 EB64: CE 71 00 ldx $7100

System routine at 0xD715 (JAPAN 1.0, EUROPE 1.0)
respectively at 0xD724 (EUROPE V1.1, JAPAN V1.2)

Purpose:

- Print the string at a given column and row via system call at `0xFF49`

Parameters:

- Register B has the number of characters in the string
- Register X has the address of the string structure
- The string structure begins with 2 bytes for 0-based col/row indices followed by the characters of the string, as counted in register B.

Re-Assembling the ROM

The assembler source code can be assembled with the A09 assembler. The source contains a few equates which can be adapted to your system:

- `ROM_CD` allows selecting the Basic ROM installed in your system at locations C000-DFFF.
- `ROM_EF` allows selecting the Basic ROM installed in your system at locations E000-FFFF.

These two can be equated to a locale of "`EUROPE`" or "`JAPAN`" and to a version of `V10` or `V11`.

This would allow for 2×2×2 = 8 combinations of ROMs, but in real life probably only the four versions already prepared in the assembler input file exist. So the first step is to identify which ROM set your HX-20 has, as detailed in the first part of this document.

My test setup consisted of a German HX-20 with V1.0 ROMs, thus I selected

```
; European, e.g. German HX-20 with Basic V1.0
ROM_CD EQU EUROPE | V10
ROM_EF EQU EUROPE | V10
```

Furthermore it is possible to add my own additional words to the FORTH dictionary and to slightly optimize the original code by setting `MH_MODS` to 1.

```
MH_EXTENSIONS  EQU 1
```

You can include the original unused trailer bytes to the end of the file (only if `MH_MODS EQU 0`) by setting

```
INCLUDE_TRAILER EQU 1
```

otherwise, the trailer is filled with $00 bytes for better visibility.

Distributed over the whole ROM I found more than 60 unused bytes. These will be included by setting

```
KEEP_FLUFF EQU 1
```

otherwise, they are removed. 60 bytes sounds like a low number, but is already sufficient for two or three short additional words.

A binary identical representation of the original ROM image is generated with the following settings:

```
ROM_CD          EQU JAPAN | V10
ROM_EF          EQU JAPAN | V10
MH_EXTENSIONS   EQU 0
KEEP_FLUFF      EQU 1
INCLUDE_TRAILER EQU 1
```

Extending the ROM

Besides adapting the ROM to the various system ROM versions, one goal of this project was to enhance the ROM. While it already contains a large number of useful words, some functions seem to be weak. These could easily be implemented in Forth, but would then consume valuable RAM.

When I disassembled the ROM, I found some unused code fragments between code and data segments and also a larger contiguous unused area at the end of the ROM. In order to make better use of the unused bytes and for learning how to extend the ROM, some additional words have been implemented. These words were added to the end of the linked chain at the end of the source file. Additional optimizations were possible by replacing a few jump instructions by branch instructions and in a few places by making use of the D register instead of using A and B separately. It became obvious, that the author had started with figForth for the 6800 and in new code regions he made use of the 6301 but some regions still had the original 6800 code.

To simplify version checking, I also added signatures for locale and version at the end of the ROM. These can be removed later to obtain more free bytes. Right now, you will find something like this at the end of the image:

```
00001FD0 93 80 44 56 7E 61 37 00 00 00 00 00 00 00 00 00 ..DV~a7.........
00001FE0 00 00 00 43 31 30 2F 45 31 30 45 2F 32 30 32 32 ...C10/E10E/2022
00001FF0 2D 4D 61 72 74 69 6E 20 48 65 70 70 65 72 6C 65 -Martin Hepperle
```

You can recognize that this ROM was created for ROMs C000-DFFF V1.0 and E000-FFFF V1.0 EUROPEAN. Any preceding zeroes are the few remaining bytes available for improvements.

The New Words

Binary Shift

The first word which I added was a simple binary shift operation for a 16-bit cell. Such a function is useful for generating bit masks and was easy enough to serve as an exercise for myself. Initially, two variants of this word have been implemented: one version `FSHIFT` was written in Forth and a second `SHIFT` was written in machine language. Both implementations were added to the ROM for testing their memory footprint and their speed. After this benchmark, only the machine language version was retained.

Forth Implementation

```
: FSHIFT
    DUP 0= IF
      DROP
    ELSE
      DUP 0< IF
        0 DO 2 / LOOP
      ELSE
        MINUS 0 DO 2* LOOP
      THEN
    THEN ;
```

The actual implementation in the ROM is the code as it would have been generated by the Forth compiler. The `IF … ELSE … THEN` constructs are replaced by conditional branch instructions. This code requires 62 bytes of ROM.

```
NFA_FSHIFT:
   FCB   $80|6              ; $80 | length of name
FSHIFT:
   FCB   "FSHIF",('T'|$80)  ; 6 characters long
LFA_FSHIFT:
```

```
   FDB   NFA_SHIFT
CFA_FSHIFT:
   FDB   DOCOL
   FDB   CFA_DUP
   FDB   CFA_ZEROEQ          ; test shift count for zero
   FDB   CFA_ZEROBRANCH      ; 0: not zero
   FDB   FSHIFT1-*           ; skip b != 0: shift

   ; case 1: shift count is zero
   FDB   CFA_DROP            ; drop shift count
   FDB   CFA_BRANCH          ; zero: no shift
   FDB   FSHIFT5-*           ; leave number
FSHIFT1:
   FDB   CFA_DUP
   FDB   CFA_ZEROLT          ; is 0 < shift count? 1: yes
   FDB   CFA_ZEROBRANCH
   FDB   FSHIFT3-*           ; positive shift count
   ; case 2: negative shift count: shift left
   FDB   CFA_NEGATE          ; make shift count positive
   FDB   CFA_ZERO
   FDB   CFA_PARDO           ; n b 0 DO
FSHIFT2:
   FDB   CFA_TWOMUL          ; * 2
   FDB   CFA_PARLOOP
   FDB   FSHIFT2-*           ; LOOP again
   FDB   CFA_BRANCH          ; done
   FDB   FSHIFT5-*

FSHIFT3:
   ; case 3: positive shift count: shift right
   FDB   CFA_ZERO
   FDB   CFA_PARDO           ; n b 0 DO
FSHIFT4:
   FDB   CFA_TWO             ;
   FDB   CFA_DIV             ; / 2
   FDB   CFA_PARLOOP
   FDB   FSHIFT4-*           ; LOOP again

FSHIFT5:
   FDB   CFA_SEMIS
```

Machine Language Implementation

Looking rather long in commented assembler mnemonics, the machine language version actually needs only 37 bytes. This does not imply that an assembler solution would always be more compact than a Forth implementation.

```
NFA_SHIFT:
   FCB   $80|5               ; $80 | length of name
SHIFT:
   FCB   "SHIF",('T'|$80)    ; 5 characters long
LFA_SHIFT:
   FDB   NFA_TWOMUL
CFA_SHIFT:
   FDB   *+2                 ; this is a machine code word
   ; drop shift count into (A)(B)
   pula                      ; get high byte
   pulb                      ; get low byte
   ; we ignore the high byte (shift count < 16)
   tstb                      ; set Z(ero) and N(egative)

   ; case 1: shift count is zero
   beq   SHIFT_DONE           ; Z is set: -> no shift, leave 2nd word on stack
;
   tsx                        ; get address of word to (X)
   ; if shift count is negative then LSHIFT else RSHIFT
   bmi   LSHIFT               ; low byte is negative -> left shift

   ; case 2: shift count is positive
RSHIFT:
   lsr   $00,x               ; shift high byte -> carry
   ror   $01,x               ; carry -> shift low byte
```

```
   decb                     ; decrement (A)
   bne   RSHIFT             ; loop until (A)=0
   bra   SHIFT_DONE         ; jmp NEXT would save 4 machine cycles
                            ; but take 1 additional byte - whoa!

   ; case 3: shift count is negative
LSHIFT:
   asl   $01,x              ; carry <- shift low byte
   rol   $00,x              ; shift high byte <- carry
   incb                     ; increment (A)
   bne   LSHIFT             ; loop until (A)=0
;
SHIFT_DONE:
   jmp   NEXT               ; done
```

For testing their relative speed, 20000 runs of a 15 bit left shift operation have been timed (`: TIMING SECONDS DMINUS 20000 0 DO 1 -15 FSHIFT DROP LOOP SECONDS D+ D. ;`). The Forth version took 166 seconds whereas the machine language version executed in 17 seconds. If we subtract the time of 3 seconds for the empty `DO…LOOP`, we obtain a speed ratio of 163/14 = 11.6. This demonstrates that coding often used words in machine language is still very useful, if execution speed is a concern. Therefore, the `FSHIFT` word was a helpful exercise but was not retained in the ROM.

A Timing Word and BCD to Binary Conversion

For timing these words, two additional words were added to the ROM. The code word `BCDBIN` converts a two-digit BCD number as returned by `SECS`, `MINS`, or, `HRS` to a binary number (e.g. `$0021` is converted into `21`).

Its Forth equivalent requires 30 bytes of ROM.

```
( BCD to binary )
: BCDBIN DUP 15 AND SWAP 16 / 10 * + ;
```

The code part of this assembler code compiles into 21 bytes of machine language.

```
CFA_BCDBIN:
   FDB   *+2                ; machine code word

   ; drop 2-digit BCD number of the form $00HL into (A)(B)
   pula                      ; high byte (unused)
   pulb                      ; low byte
   pshb                      ; save for the low nibble

   andb   #$F0               ; mask H nibble

   ; 10*H/16 = 5*H/8 = 4*H/8 + 1*H/8 = H/2 + H/8
   lsrb                      ; H/2
   tba                       ; save H/2 in (A)
   lsrb                      ; H/4
   lsrb                      ; H/8
   aba                       ; (A) = H/2 + H/8 == high nibble in binary

   pulb                      ; get back for L nibble
   andb   #$0F               ; mask lower nibble
   aba                       ; and add it to binary high nibble in (A)

   tab                      ; (A) to (B)
   clra                     ; clear (A)
   jmp   NEXT_PSH_D         ; push D and then NEXT
```

The Forth word `SECONDS` makes use of this word to return the current seconds since midnight as a double word.

```
( get current time in seconds )
: SECONDS TIME@ HRS BCDBIN 3600 U*
               MINS BCDBIN 60 U* D+
               SECS BCDBIN S->D D+ ;
```

Using the RS232C Interface

The HX-20 has two serial interfaces. Here we use the RS232C interface, which is good for operation at up to 4800 baud. The interface must be powered up before it can be used and should be powered down before longer periods of inactivity to conserve battery power.

A design quirk of the HX-20 is that the power up sequence transmits one garbled character. Therefore it is recommended to power up the RS232C port before switching the printer or terminal on.

The new word `RSPWR` must be used to power the port on or off. It also sets the baud rate to 4800, the word length to 8 bits and the number of stop bits to 1. After the port has been powered on, 8-bit characters can be sent with the word `RSPUT`. Similarly, `RSGET` can be used to read characters from the RS232 receive buffer. The size of this buffer has been set to 255 bytes.

A simple word for receiving and displaying one line of text could look like so:

```
: RS232LINE
 BEGIN
  RSGET
  IF
   DUP 10 =          ( line feed ? )
   DUP 26 =          ( Ctrl-Z end of file ? )
   OR
   0=
   IF
    EMIT 0           ( continue )
   ELSE
    EMIT 1           ( exit BEGIN ... UNTIL )
   THEN
  ELSE
   0                 ( continue )
  THEN
 UNTIL ;

( application )

1 RSPWR
RS232LINE
0 RSPWR
```

Depending on the `PRINT` settings, the received characters can be echoed back or copied to the printer.

To minimize the code size, no error checking or handshaking is performed. Usually this is no problem – however, if your output device is slow or has a small input buffer, you must insert appropriate delays.

Instead of a printer, you can also connect a terminal or a terminal program to collect the output. When reading data from the terminal, it may be necessary to specify an inter-character spacing of a few milliseconds to achieve smooth data communication without handshaking.

An alternative would be to modify the initialization word in the code to use CTS/RTS handshaking.

Improved Print Flag

The existing word `PRINT` has been enhanced so that it can now copy output to the RS232C port.

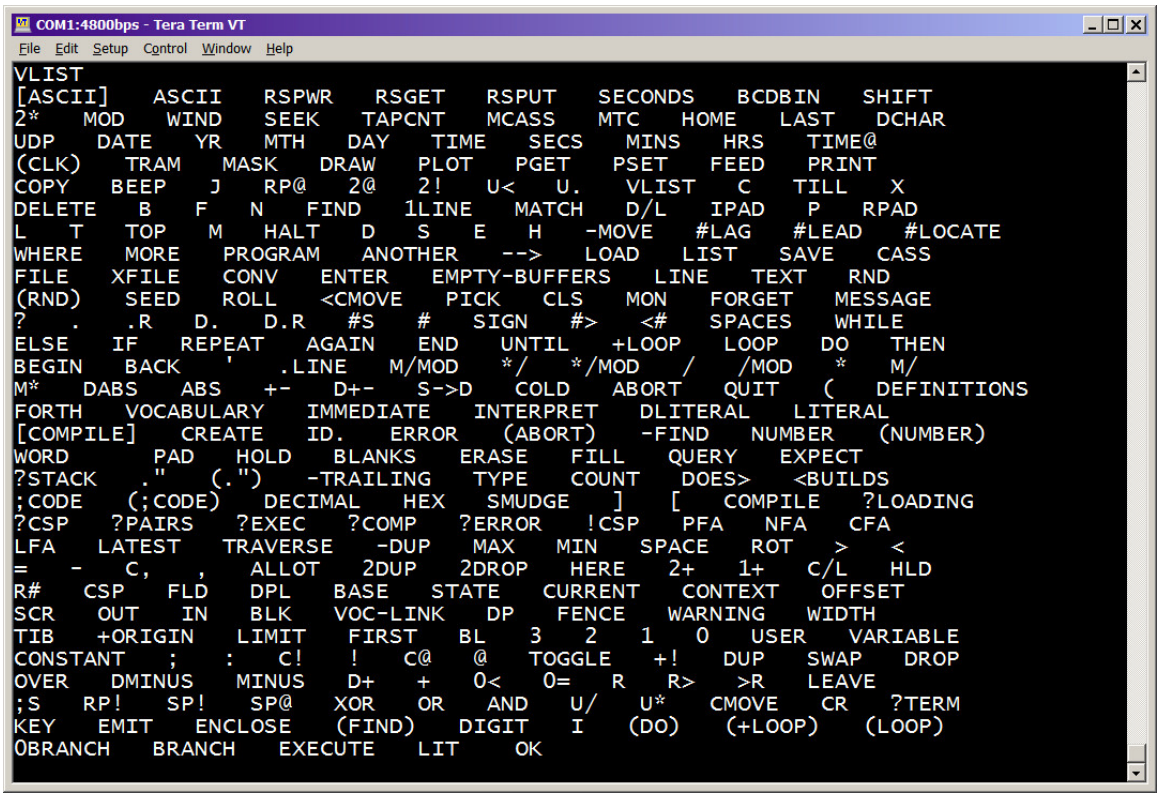

Figure 1: **The enhanced PRINT word allows copying the standard output to the RS232C port. The top row lists the words added to the ROM.**

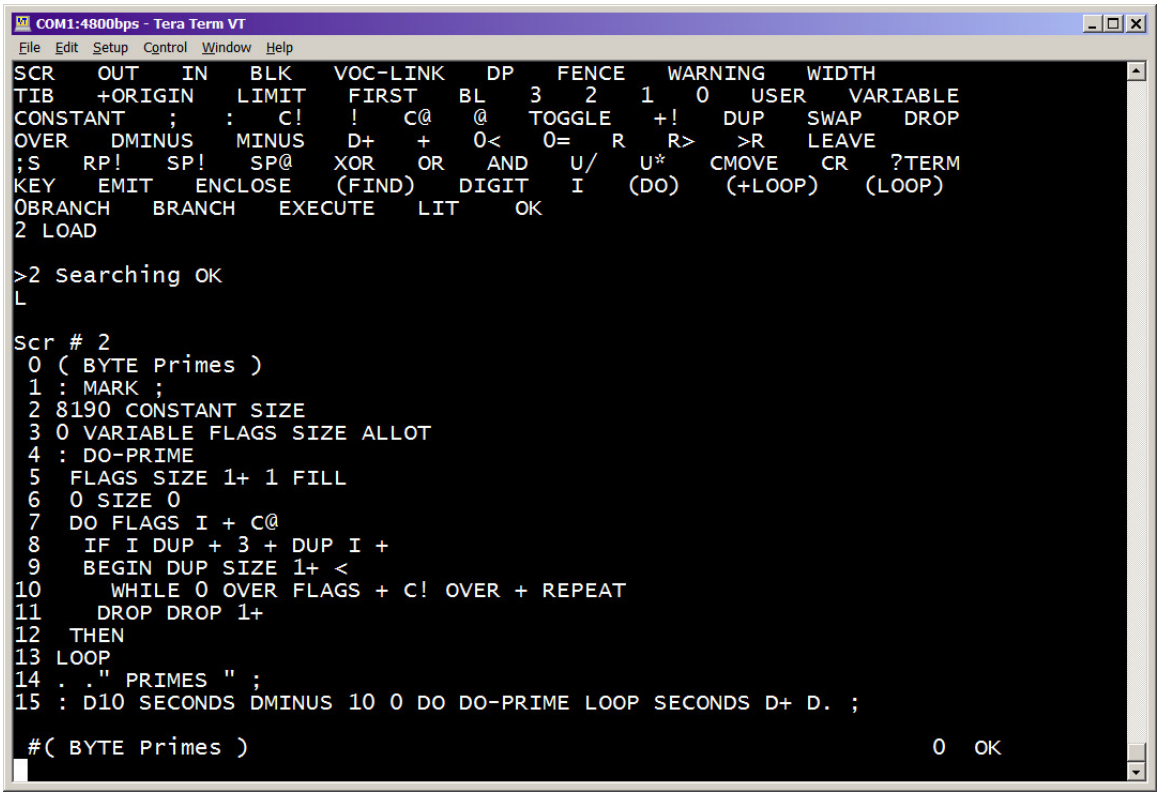

Figure 2: **Sending the output to a terminal makes editing on the HX-20 much more convenient. The row following line 15 shows the PAD with the currently edited line (in this case line 0) with the editing cursor # (currently before the first character of this line). Cursor editing commands will change this line of the current screen.**

Summary of the new or enhanced Words

```
SHIFT       (n1 b --- n2)
```

Shift n1 by b bits left (–) or right (+). b should be in +/-[0...15].

```
2/          (n1 --- n2)
```

Divide the number n1 by 2.

```
BCDBIN      (n1 --- n2)
```

Convert the positive two-digit BCD number n1 to a binary number n2. n1 should be in [$0000...$0099].

```
SECONDS     (--- ud)
```

Leave the unsigned double number ud with the seconds elapsed since midnight. ud can be within [0...86399].

```
RSPWR       (f ---)
```

Apply (f=1) or remove (f=0) power from the RS232C port.
Set the RS232C transmission parameters to 4800 baud, 8 data bits, 1 stop bit, no parity, no handshaking. These settings are currently hardwired into the ROM code.
This word uses the cassette buffer for a 260 byte receive buffer. Therefore no cassette operations should be performed while the RS232C port is open. Close it with a `0 RSPWR`, also to reduce power consumption.
Due to a design flaw, the HX-20 will always output one spurious character when powering the RS232C port on. This may lead to e.g. a form feed on a printer or worse if you are controlling a nuclear power plant with your HX-20.

```
RSGET       (--- b f=1), (--- f=0)
```

Try to read one byte b from the RS232C port buffer. Flag f=1 if there was a byte available, f=0 otherwise. The port must have been powered on first by `1 RSPWR`.

```
RSPUT       (b ---)
```

Output the byte b to the RS232C port. The byte is output with all its 8 bits. The port must have been powered on first by `1 RSPWR`.

```
PRINT       (f ---)
```

Select the path for console output:

- f=0: no output – not too useful,
- f=1: output to the LCD screen (default),
- f=2: output to the internal printer,
- f=4: output to the RS232C port,

You can combine these bits, for example output to the LCD screen and to the RS232C port would require a `1 4 + PRINT`.
Note that bit 7 of this console output is always stripped, i.e. only ASCII characters in the range of 0…127 will be output. The RS232C port has to be powered on, otherwise no output will take place.

```
DEPTH       (--- n)
```

Return the number of cells currently on the computational stack. After a cold start, `DEPTH` returns `0`.

```
ASCII       (--- b)
```

Return the ASCII code of the first character of the next word (usually a single character). Used in the form `ASCII c` . For example, `ASCII A` returns `65`.

```
[ASCII]     (--- b)
```

Compile the ASCII code of the first character of the next word (usually a single character) into a definition. Used in the form `ASCII c` . For example, `ASCII A` compiles `65`.

```
KEY         (--- n)
```

This word has been enhanced. The original implementation returned $FE for all PF keys, neglecting the 2-byte sequence sent by these keys. The improved version returns the values listed in the table below.

Key	Normal		Shifted	
PF1	-271	$FEF1	-266	$FEF6
PF2	-270	$FEF2	-265	$FEF7
PF3	-269	$FEF3	-264	$FEF8
PF4	-268	$FEF4	-263	$FEF9
PF5	-267	$FEF5	-262	$FEFA

Table 6: PF keys and the codes returned by the KEY word.

If you test the result for a negative number, you can easily identify the PF keys.
Unfortunately it is not possible to read the cursor keys using KEY. If you want to check the status of all keys, you can analyze the result of the last scan of the keyboard matrix. The result is stored in 10 bytes, starting at address $140. There are also two additional tables which store the previous scan result, so that one could also detect make and release actions.
The Technical Manual contains tables which I found hard to read and difficult to interpret. Therefore I have rewritten the scan table and the associated keys in a, what I believe, more readable form. Note that the physical keys are of course the same, but the labels will depend on the localization of your device. I have listed the US variant and the German layout only.

Address	Key Label							
$145	7	6	5	4	3	2	1	0
$146	/	.	_	,	;	:	9	8
$147	G	F	e	D	C	B	A	@
$148	O	N	M	L	K	J	I	H
$149	W	V	U	T	S	R	Q	P
$14A	←	→	\	]	[	Z	Y	X
$14B	CAPS	GRAPH	NUM	n.a.	n.a.	TAB	SPACE	RETURN
$14C	n.a.	n.a.	MENU	DEL	PAUSE	BREAK	SCRN	CLR
$14D	n.a.	n.a.	PAPER	PF5	PF4	PF3	PF2	PF1
$14E	PRINT	CTRL	SHIFT	n.a.	DIP4	DIP3	DIP2	DIP1
Mask	*0x80*	*0x40*	*0x20*	*0x10*	*0x08*	*0x04*	*0x02*	*0x01*

Table 7: Key scan table with the US key caps.

Address	Key Label							
$145	7	6	5	4	3	2	1	0
$146	-	.	ß	,	Ö	Ä	9	8
$147	G	F	e	D	C	B	A	Ü
$148	O	N	M	L	K	J	I	H
$149	W	V	U	T	S	R	Q	P
$14A	←	→	^	#	+	Y	Z	X
$14B	CAPS	GRAPH	<	n.a.	n.a.	TAB	SPACE	RETURN
$14C	n.a.	n.a.	MENU	DEL	PAUSE	BREAK	SCRN	CLR
$14D	n.a.	n.a.	PAPER	PF5	PF4	PF3	PF2	PF1
$14E	PRINT	CTRL	SHIFT	n.a.	DIP4	DIP3	DIP2	DIP1
Mask	*0x80*	*0x40*	*0x20*	*0x10*	*0x08*	*0x04*	*0x02*	*0x01*

Table 8: Key scan table with the German key caps.

In order to test for the cursor control keys you could use something like this code fragment:

```
HEX BEGIN 14A C@ DUP   80 = IF ." LEFT " 0 THEN   40 = IF ." RIGHT " 0 THEN  ( cursor keys )
          14B C@        1 = IF 1 THEN                                         ( Return )
    UNTIL
```

The infamous BYTE Benchmark

Again, the classical BYTE Prime benchmark was entered into the screen editor. The algorithm was implemented as printed in BYTE Magazine, only with small error corrections.

```
 0 ( BYTE Sieve benchmark )
 1 8190 CONSTANT SIZE
 2 0 VARIABLE FLAGS SIZE 1+ ALLOT
 3 : DO-PRIME
 4  FLAGS SIZE 1+ FILL
 5  0 SIZE 0
 6  DO FLAGS 1+ C@
 7    IF I DUP + 3 +DUP I +
 8      BEGIN DUP SIZE 1+ <
 9      WHILE 0 OVER FLAGS + C! OVER + REPEAT
10      DROP DROP 1+
11    THEN
12  LOOP
13 . ." PRIMES" ;
14
15 : D10 SECONDS DMINUS 10 0 DO DO-PRIME LOOP SECONDS D+ D. ;
```

The word D10 executes to prime algorithm 10 times within 229 seconds. This is about 20 times as fast as in BASIC, but about 14 times slower than a pure assembler version. Thus we see that Forth can be a very efficient solution in terms of development and execution time.

Computer	Year	CPU Type and Speed	Programming Language	Time
HX-20	1982	6301 @ 0.614 MHz	BASIC	4050 s
HX-20	1982	6301 @ 0.614 MHz	Assembler	17 s

HX-20	1982	6301 @ 0.614 MHz	Forth	229 s
TI-99/4	1981	TMS 9900 @ 3.0 MHz	TI-BASIC	3960 s
PET	1977	6502 @ 1.0 MHz	BASIC	3180 s
Apple][	1977	6502 @ 1.02 MHz	Applesoft BASIC	2806 s
HP-85	1980	Capricorn @ 625 kHz	BASIC	3084 s
HP-85	1980	Capricorn @ 625 kHz	Assembler	21 s
TRS-80/II	1977	Z-80 @ 1.77 MHz	MBASIC	2250 s
IBM PC	1981	8088 @ 4.77 MHz	BASICA	1990s

Table 9: Some execution times for the BYTE benchmark.

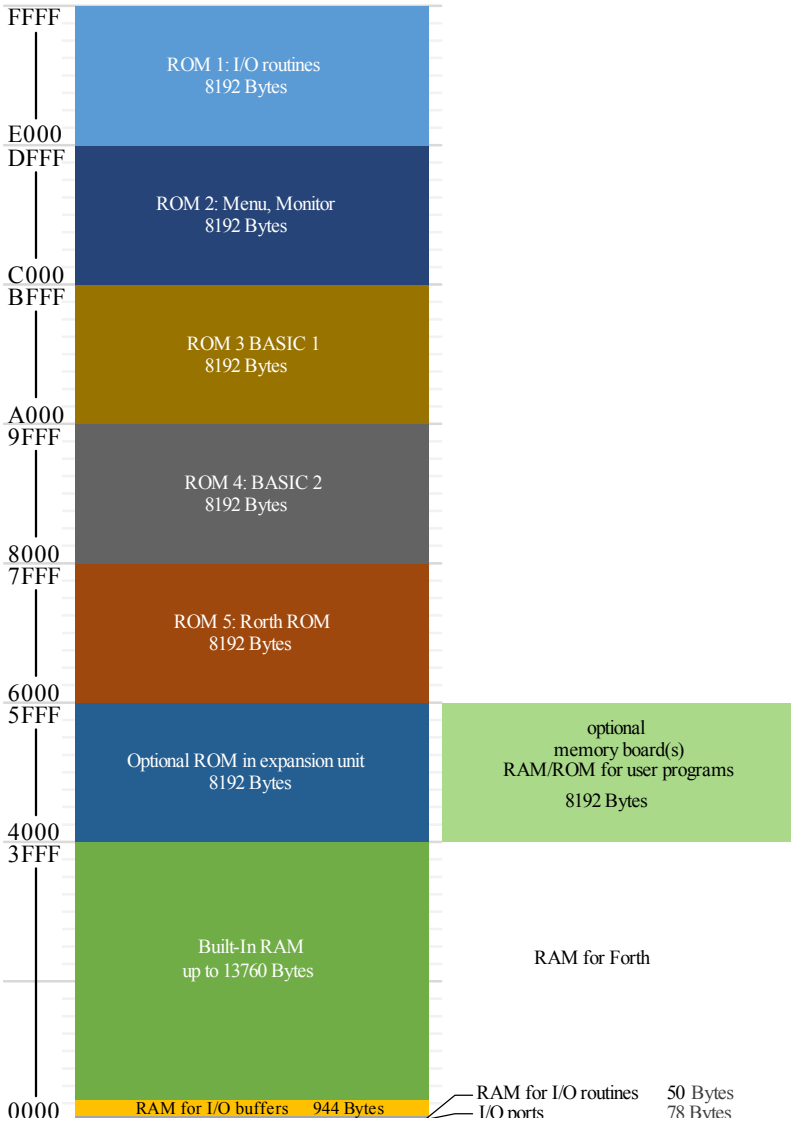

Figure 3: Global memory map of a standard HX-20 system with the Forth ROM.

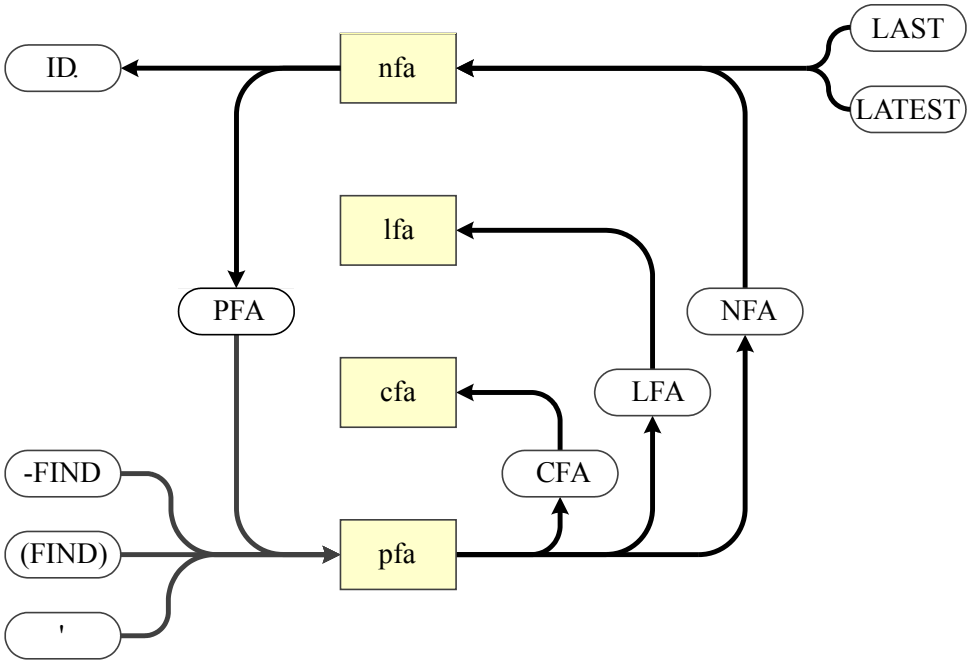

Figure 4: Addresses and related words used in figForth.

		Address		
		HEX	DEC	Comments
less than 12287 bytes		3FFF	16383	
68 bytes above HERE	PAD	1044	4164	sliding above HERE
				Dictionary grows upwards
	DP @	1000	4096	Directory Pointer points to HERE
	HERE	1000	4096	
	FENCE	1000	4096	Lower Limit of user Dictionary
1024 bytes	LIMIT	0B30	2864	I/O Buffer/Header
	FIRST	0710	1808	Editor Screen 16 lines x 64 characters
256 bytes	SP@ = S0	06FE	1790	Parameter Stack grows downwards
		05FF	1535	
191 bytes	RP@ = R0	05FE	1534	Return Stack grows downwards
		0540	1344	
64 bytes		053F	1343	Terminal Input Buffer
	TIB @	0500	1280	64 bytes long up to 53F
User Variables		04E8	1256	User Variables MASK
		04BA	1210	User Variables TIB
				System Variables
		0000	0	

Figure 5: Memory map of the HX-20 figForth.

Address	User Variable	Offset DEC	Length	Contents HEX	Contents DEC
4E8	MASK	56	2	7F	127
4E6		54	2	0	0

4E4		52	2	0	0
4E2		50	2	0	0
4E0	HLD	48	2	1040	4160
4DE	R#	46	2	FFE0	65504
4DC	CSP	44	2	6FE	1790
4DA	FLD	42	2	0	0
4D8	DPL	40	2	FFFF	65535
4D6	BASE	38	2	10	16
4D4	STATE	36	2	0	0
4D2	CURRENT	34	2	D0E	3342
4D0	CONTEXT	32	2	D0E	3342
4CE	OFFSET	30	2	0	0
4CC	SCR	28	2	1	1
4CA	OUT	26	2	312	786
4C8	IN	24	2	5	5
4C6	BLK	22	2	0	0
4C4	VOC-LINK	20	2	0D0E	3342
4C2	DP	18	2	1000	4096
4C0	FENCE	16	2	1000	4096
4BE	WARNING	14	2	0	0
4BC	WIDTH	12	2	1F	31
4BA	TIB	10	2	500	1280
4B8	R0	8	2	5FE	1534
4B6	S0	6	2	6FE	1790

Figure 6: Map of the user variables in the HX-20 figForth.