JBit E1 (1)

Preface

The E1 series will show you how to write a complete application with JBit. While the application is trivial by today's standards, you will write it from scratch in Assembly, gaining some insight about the inner workings of many simple everyday devices. Moreover, you will be able to use the techniques presented here to write more complex applications.

I assume you have a good understanding of the 6502 demos included in the standard JBit distribution and some experience in writing and testing short snippets of code with JBit.

Usage

The application is a simulation of a digital clock.

When the application starts, 12AM is shown. You can change this value by pressing `9` and confirm it by pressing `3`. You can then change the minutes: first the most significant digit and then the least significant digit. When you confirm the the least significant digit, the dots start blinking and the clock starts keeping the time. You can switch between the HH:MM view and the SS view by pressing `9` or change the time again by pressing `3`.

Program Layout

The first step is to choose the layout of the program. With the current version of JBit this is an important decision because you cannot change it later. Estimating the layout of a program can be difficult. Fortunately, you can use more pages than you need and, as long as you keep the unused pages filled with 0s, the additional space has very limited overhead.

Here is the layout that we will use:

$\mathbf{3_C}$	$\mathbf{4_C}$	$\mathbf{5_C}$	$\mathbf{6_C}$	$\mathbf{7_C}$	$\mathbf{8_D}$	$\mathbf{9_D}$
T	*	*	*	E	*	E

The program itself uses 3 pages of code (4-6) and 1 page of data (8). We will use page 3 as the main test area and will reserve two additional pages (7 and 9) for further experimentations.

Subroutines

The main technique to be able to write large programs is to code and test them a piece at a time. A *subroutine* is an independent block of code that can be written and tested in isolation.

The 6502 provides two instructions to help you in writing subroutines: JSR (Jump to SubRoutine) and RTS (ReTurn from Subroutine). When you want to use a subroutine (a.k.a. *calling* the subroutine) you use JSR. JSR is similar to JMP, but saves the current PC. When the subroutine has done its job, it uses RTS to go back to the saved PC.

A subroutine can in turn call another subroutine (a.k.a. *nesting*). The PC is saved on page 1, so you can nest at most 128 calls (remember that the PC is 2 bytes wide). The offset of the next free cell on page 1 is stored in the Stack register. The Stack register is initialized to 255, decremented by 2 on JSR and incremented by 2 on RTS.

Here is an example of how page 1 would look like after two nested subroutine calls:

...	**251**	252	253	254	255
...	-	20	3	90	3

Tables

We might not be sure yet how to write the whole application, but sooner or later we will need to be able to write and erase three strings (AM, PM and :). We will start from here. Beginning from the most basic pieces that you are very likely to end up using later on is called *bottom-up* programming.

The *ciao* demo writes a string on the display using a series of LDA/STA pairs and thus takes 5 bytes of code to write a single character. To make the code shorter, we will use a data table:

	$\mathbf{D_0}$	$\mathbf{C_0}$	$\mathbf{D_1}$	$\mathbf{C_1}$
AM	32	A	33	M
PM	35	P	36	M
:	14	:	24	

Each string is stored as a pair of characters and each character is stored as a pair of bytes: first its displacement relative to CONVIDEO and then its ASCII code. Note that the single-character string ":" is stored as the two-character string ": " to make the table simpler to use. To erase the string we will only need to use the displacements.

Listing

```
   * 3:0 *         * CODE *
 0 LDX #0          TEST           3:0
 2 LDA #1          VLEDDRAW       4:0
 4 JSR 4:0         * DATA *
 7 LDX #8          CONVIDEO      2:40
 9 LDA #1          TAB-L-D0       8:0
11 JSR 4:0         TAB-L-C0       8:1
14 BRK             TAB-L-D1       8:2
   * 4:0 *         TAB-L-C1       8:3
 0 BNE 4:17        * CONSTANTS *
 2 LDA #32         ASC-SPC         32
 4 LDY 8:0,X       DSP-AM           0
 7 STA 2:40,Y      DSP-DOTS         8
10 LDY 8:2,X       DSP-PM           4
13 STA 2:40,Y
16 RTS
17 LDY 8:0,X
20 LDA 8:1,X
23 STA 2:40,Y
26 LDY 8:2,X
29 LDA 8:3,X
32 STA 2:40,Y
35 RTS
```

Type In

```
C 3:0
000: 162 000 169 001 032 000 004 162
008: 008 169 001 032 000 004 000 000

C 4:0
000: 208 015 169 032 188 000 008 153
008: 040 002 188 002 008 153 040 002
016: 096 188 000 008 189 001 008 153
024: 040 002 188 002 008 189 003 008
032: 153 040 002 096 000 000 000 000

D 8:0
000: 032 065 033 077 035 080 036 077
008: 014 058 024 032 000 000 000 000
```

Typing Instructions

Start JBit, select Editor, select 5 pages of code and 2 pages of data, select OK to confirm, select Save, enter a name for the program and select OK.

Type in the bytes. Check often the address of the current cell to make sure that you don't skip any bytes. While typing the code pages, press # once in a while to check that what you have typed matches the listing. Use the GoTo command to go directly to a particular location.

While I personally never experienced any data loss, you might want to save often. Occasionally, go to the Store and make a backup copy of the program.

Comment

The subroutine VLEDDRAW turns ON or OFF a Virtual LED on the display. It expects the register X to contain which Virtual LED should be updated. To make the code shorter the displacement of the row relative to the start of the table is used. VLEDDRAW uses the result of the last operation as the status of the Virtual LED (zero means OFF and non zero means ON).

The test code turns ON the AM and DOTS Virtual LEDs.

Suggestions

There is no way you can learn how to write your own programs by just reading. This series assumes you are willing to spend some time tinkering with the code.

Change the test code to be able to test turning a Virtual LED OFF too. For example, you can make the dots blink. Leverage the fact that writing 0 to 2:18 causes the CPU to be suspended until the current frame has been drawn. The test code will be replaced anyway, so you don't need to revert to the original version.

Once you have a richer test case, do some step-by-step executions to make sure you understand how VLEDDRAW works.

When everything is clear, try to come up with a different way to do what VLEDDRAW does. Leave VLEDDRAW as it is (you will need it later). Use page 7 for your code and page 9 for your data and change the JSR instructions to switch between the original subroutine and yours at any time.

JBit E1 (2)

Binary Notation

A *bit* is a binary digit (i.e. 0 or 1). A *byte* is a pattern of 8 bits (e.g. 00011101). Bits in a byte are identified by their position, starting from 0 for the rightmost one (e.g. the 5^{th} bit from the right is called bit 4).

Bits and bytes have no meaning by themselves; it is the programmer who gives meaning to them. 01000001 could mean "red" in a piece of code and "A" in another piece of code. However, a lot of operations (e.g. INC) and addressing modes (e.g. n:n,X) of the 6502 only make sense if bytes are interpreted as integers ranging from 0 to 255. Furthermore, small integers are familiar and useful. This is why JBit shows bytes in this way.

So, how 01000001 and 65 relate to each other? I spare you the theory (hint: $...b_2b_1b_0$ is $1b_0 + 2b_1 + 4b_2 + ...$, just like 237 is $1 \times 7 + 10 \times 3 + 100 \times 2$) and I only show you a couple of practical conversion techniques using a support table:

Decimal to Binary. Take the byte (195), find the biggest multiple-of-16 that is not greater than the byte (192, i.e. 1100), subtract the multiple-of-16 from the byte (195 - 192 = 3, i.e. 0011) and join the two results (11000011).

Binary to Decimal. Take the byte (10100110), split it in two halves (1010 and 0110), use the multiple-of-16 column to convert the first half (160), use the counting column to convert the second half (6) and add the two results (160 + 6 = 166).

0000	0	0	1000	8	128
0001	1	16	1001	9	144
0010	2	32	1010	10	160
0011	3	48	1011	11	176
0100	4	64	1100	12	192
0101	5	80	1101	13	208
0110	6	96	1110	14	224
0111	7	112	1111	15	240

Boolean Operations

Treating bytes as patterns of bits leads to an intuitive understanding of the so called *boolean operations*:

```
01010101 AND   01010101 OR   01010101 XOR
00110011 =     00110011 =    00110011 =
--------       --------      --------
00010001       01110111      01100110
```

These operations are carried out on two bytes considering their corresponding bits in isolation. The result of AND is 1 if both bits are 1, otherwise is 0. The result of OR is 1 if at least one bit is 1, otherwise is 0. The result of XOR is 1 if the bits are different from each other, otherwise is 0.

The 6502 opcodes for these operations are: AND for AND, ORA (OR with Accumulator) for OR and EOR (Exclusive OR) for XOR. All of them operate on the Accumulator.

As an example, to compute the value of the Accumulator after this snippet of code:

```
LDA #58
ORA #12
```

convert the bytes to binary notation, do the operation and convert the result to decimal notation:

```
00111010  (58)  OR
00001100  (12)  =
--------------
00111110  (62)
```

Bitmasks

One of the most common use of the boolean operations is to pack (unpack) small pieces of information into (from) a byte.

Consider how bits 2 and 3 of the first operand are affected by the following operations:

```
00111010  OR   00001100  (12)  =  00111110
00111010  XOR  00001100  (12)  =  00110110
00111010  AND  11110011 (243)  =  00110010
```

The second operand is called *bitmask*. OR can be used to set bits to 1, XOR to invert them (1 becomes 0 and 0 becomes 1) and AND to reset them to 0.

The decimal notations of single-bit bitmasks are usually known by heart by assembly programmers (1, 2, 4, 8, 16, 32, 64 and 128). Inverting a byte (i.e. inverting all its bits) by hand to compute a bitmask for the AND operation can be done by subtracting it from 255.

Consider also:

```
01110100  AND  00001000   (8)  =  00000000
01111100  AND  00001000   (8)  =  00001000
```

AND can be used to test a single bit using BEQ and BNE.

Listing

```
   * 3:0 *          * CODE *
 0 LDA #8           TEST           3:0
 2 STA 2:17         VLEDDRAW       4:0
 5 LDA #2           VLDSDRAW      4:36
 7 STA 80           * DATA *
 9 LDA 80           FRMFPS         2:17
11 EOR #4           FRMDRAW        2:18
13 STA 80           * ZERO PAGE *
15 JSR 4:36         VLDS-MSK         40
18 LDA #0           TEST-TMP         80
20 STA 2:18         * CONSTANTS *
23 JMP 3:9          DSP-AM            0
   * 4:36 *         DSP-DOTS          8
36 STA 40           DSP-PM            4
38 LDX #0           MSK-AM            1
40 AND #1           MSK-DOTS          4
42 JSR 4:0          MSK-PM            2
45 LDX #4           TWO-FPS           8
47 LDA 40
49 AND #2
51 JSR 4:0
54 LDX #8
56 LDA 40
58 AND #4
60 JMP 4:0
```

Type In

```
C 3:0
000: 169 008 141 017 002 169 002 133
008: 080 165 080 073 004 133 080 032
016: 036 004 169 000 141 018 002 076
024: 009 003 000 000 000 000 000 000

C 4:32
032: 153 040 002 096 133 040 162 000
040: 041 001 032 000 004 162 004 165
048: 040 041 002 032 000 004 162 008
056: 165 040 041 004 076 000 004 000
```

Typing Notes

Start JBit, select Store, load the program and type in the new values. If you end up with page 3 still having some old values, you can clear them by repeatedly pressing * in EDT mode.

Despite what we are doing here, it is usually better to leave a few BRK (0) instructions between subroutines to allow for future changes.

Comment

VLDSDRAW expects a bitmask in the Accumulator and turns ON or OFF the individual Virtual LEDs accordingly. First the bitmask is saved, then every bit is isolated and tested in turn. Now you can see why I have chosen to test if the result is not zero in VLEDDRAW; in bottom-up programming thinking in advance about how a subroutine will be used can be helpful. Note the JMP at the end of the subroutine as a faster and shorter replacement for:

```
60 JSR 4:0
63 RTS
```

The test code is composed of a setup section followed by an infinite loop. In the setup section the refresh rate is set to 2 frames per seconds (remember that in FRMFPS goes the FPS multiplied by 4) and a temporary memory location is initialized with the status of the Virtual LEDs. In every iteration of the loop the status of the DOTS Virtual LED is inverted, the Virtual LEDs are drawn and the display is updated (remember that writing 0 to FRMDRAW suspends the CPU until the next frame is drawn).

Puzzles

Binary Notation. Look at these numbers:

```
00000011   (3)        00110000  (48)
00000110   (6)        01100000  (96)
00001100  (12)        11000000 (192)
00011000  (24)
```

How do the binary numbers change? How do the decimal numbers change? How could you divide 11011000 by 4 without converting it to the decimal notation?

Another boolean operation is NOT. It inverts every bit of a byte (e.g. NOT of 01010111 is 10101000). How could you simulate it on a 6502?

Modulo Arithmetic. Everyone knows that 8 hours after 9AM is 5PM; after all, 9 + 8 = 17 and 17 - 12 = 5. What day of the month is 4 weeks after September 20? What happens when you use ADC to add 102 and 204 together? What time is it 26 minutes after 8:59? How do you find out? 26 + 59 - 60? Or just 26 - 1, saving one step? What happens when you use ADC to add 102 and 255 together?

JBit E1 (3)

Shifting

The 6502 provides some operations to shift every bit of a byte by one place. The operations are: ASL (Arithmetic Shift Left), LSR (Logical Shift Right), ROL (ROtate Left) and ROR (ROtate Right). All of them can operate either on the Accumulator or on a Memory Cell. The bit that goes out of the byte goes to the Carry. ASL and LSR fill the bit left empty with 0. ROL and ROR fill the bit left empty with the value of the Carry at the beginning of the operation.

Let's assume that the Accumulator or a Memory Cell contains the byte $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$ and the Carry contains the bit b_c, here is what happens:

Op.	**Acc./Mem.**	**C**
ASL	$b_6, b_5, b_4, b_3, b_2, b_1, b_0, 0$	b_7
LSR	$0, b_7, b_6, b_5, b_4, b_3, b_2, b_1$	b_0
ROL	$b_6, b_5, b_4, b_3, b_2, b_1, b_0, b_c$	b_7
ROR	$b_c, b_7, b_6, b_5, b_4, b_3, b_2, b_1$	b_0

Indirect Addressing

Let's assume that Y contains 1, X contains 2 and page 0 looks like this:

...	**40**	**41**	**42**	**43**	...
...	10	6	20	7	...

Now consider the following example:

```
LDA (40),Y
STA (40,X)
JMP (0:40)
```

The expression inside the parenthesis is used to point to two memory cells containing an address; first the offset and then the page.

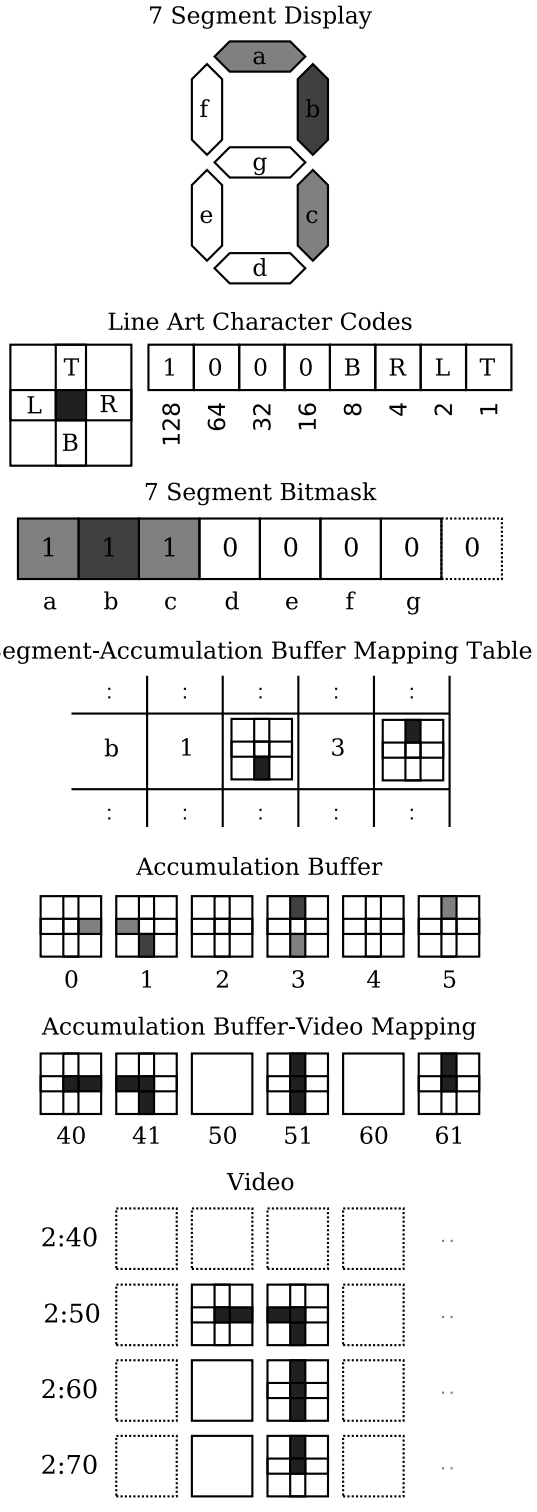
In the example above, LDA would load the Accumulator from 6:11. STA would store the Accumulator in 7:20. JMP would jump to 6:10.

Both the (n),Y and the (n,X) addressing modes are widely available, but note that (n),X and (n,Y) are *not* available. The (n:n) addressing mode is only available with the JMP instruction.

Indirect addressing is not used very often, but it can be very handy. We will use it a couple of times.

7 Segment Display

I don't want to subject you to my "English" more than necessary, so here is a figure highlighting the contribution of the segment b in drawing the digit 7 (bitmask 224) at offset 11:



Listing

```
   * 3:0 *
 0 LDA #238
 2 LDX #4
 4 JSR 4:63
 7 BRK
   * 4:63 *
63 STX 47
65 STA 49
67 LDA #0
69 TAY
70 LDX #6
72 STA 40,X
74 DEX
75 BNE 4:72
77 ASL 49
79 BCC 4:101
81 LDX 8:12,Y
84 LDA 8:13,Y
87 ORA 41,X
89 STA 41,X
91 LDX 8:14,Y
94 LDA 8:15,Y
97 ORA 41,X
99 STA 41,X
101 INY
102 INY
103 INY
104 INY
105 CPY #28
107 BNE 4:77
109 LDA #2
111 STA 48
113 LDX #0
115 LDA 41,X
117 BEQ 4:123
119 ORA #128
121 BNE 4:125
123 LDA #32
125 LDY 8:40,X
128 STA (47),Y
130 INX
131 CPX #6
133 BNE 4:115
135 RTS
```

```
* CODE *
TEST          3:0
SSEGDRAW     4:63
* DATA *
TAB-S-D0     8:12
TAB-S-M0     8:13
TAB-S-D1     8:14
TAB-S-M1     8:15
TAB-AV-D     8:40
* ZERO PAGE *
ACCBUF         41
VPTR-LO        47
VPTR-HI        48
MASK           49
* CONSTANTS *
ASC-SPC        32
A-DSP           4
A-MSK         238
CHR-DOT       128
IO-HI           2
```

Type In

```
C 3:0
000: 169 238 162 004 032 063 004 000

C 4:56
056: 165 040 041 004 076 000 004 134
064: 047 133 049 169 000 168 162 006
072: 149 040 202 208 251 006 049 144
080: 020 190 012 008 185 013 008 021
088: 041 149 041 190 014 008 185 015
096: 008 021 041 149 041 200 200 200
104: 200 192 028 208 224 169 002 133
112: 048 162 000 181 041 240 004 009
120: 128 208 002 169 032 188 040 008
128: 145 047 232 224 006 208 236 096

D 8:8
008: 014 058 024 032 000 004 001 002
016: 001 008 003 001 003 008 005 001
024: 004 004 005 002 002 008 004 001
032: 000 008 002 001 002 004 003 002
040: 040 041 050 051 060 061 000 000
```

Comment

The subroutine SSEGDRAW draws a Virtual 7 Segment Display. It expects the register X to contain the displacement of the top-left corner relative to the origin of the video and the Accumulator to contain the bitmask for the LEDs. First the displacement and the bitmask are saved and the accumulation buffer is cleared. Then the code from 4:77 to 4:104 is repeated 7 times ($4 \times 7 = 28$). Every bit of the bitmask is isolated (ASL) and tested (BCC); if the bit is set, two cells of the accumulation buffer are "painted" (ORA) according to the TAB-S table. After this loop the accumulation buffer is mapped to the screen transforming every cell to the appropriate character code. If the cell is 0 an ASCII space is drawn, otherwise the bit 7 is set to get the right JBit character code. Note that the line art character codes are not part of the Latin1 character set.

The test code draws a stylized "A" on the display.

SSEGDRAW is perhaps the most complex subroutine of the whole application, so don't worry if you don't understand it at first. Step-by-step execution is not going to help you much if you don't get the main idea from the figure on the other side of this sheet. You can complete the rest of the series without understanding how SSEGDRAW works, so take your time. Just make sure you know how to use it by playing with the test code.

JBit E1 (4)

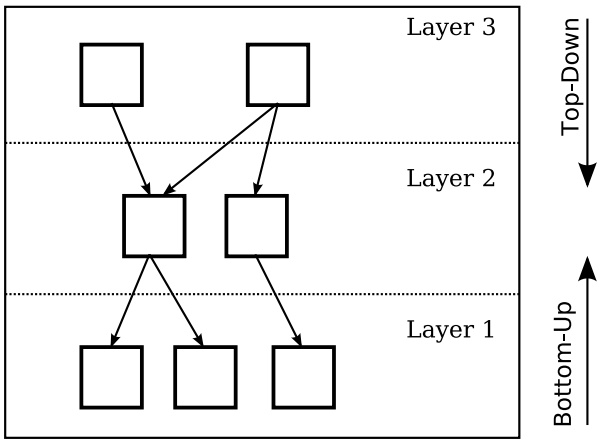
Programming Techniques

Bottom-Up Programming. Starting writing a bunch of subroutines cannot lead anywhere, right? Well, it turns out that this is often a good way to start. When you first reason about a program you usually find out that the concepts you are using (numbers, score, enemy, wall, etc...) are not the ones available to you (in Assembly basically bits and bytes). These concepts can suggest you useful building blocks.

Top-Down Programming. Another approach is to start with the "big picture". For example, you could start with a program calling some empty subroutines (e.g. intro, play, win and game over) and code them later. In practice, alternating between Bottom-Up and Top-Down can be quite effective.

Information Hiding. The less information you need to use a building block (a.k.a. *interface*), the better. Once something works, you should be able to forget how it works (a.k.a. *implementation*). This allows you to focus on part of the program. It also allows you to define an interface, build a simplified implementation, see if it makes sense and then rewrite it later if needed or throw it away otherwise. If you keep the same interface, you don't need to rewrite the code using it. Cheap and fast prototyping with toy building blocks is essential to foster creativity.

Abstraction Layers. Another way to limit the details you have to consider is to group the building blocks in layers; each layer only interacting with the layer below and the layer above. This fits well with the bottom-up and top-down approach, where the lowest layers are the nearest to the machine.



Hardware Abstraction Layer (1)

We will come up with an abstraction layer that mimics the hardware of a fictional digital clock. This apparently contrived approach will allow you to better understand JBit and how it compares to real devices. It will also give you some insight on how some real devices are programmed.

On this sheet we will start by analyzing some of the most subtle differences.

JBit has only RAM (i.e. every memory cell can be read and written). Part of the memory is initialized using the program as a template, the rest is filled with 0s. Once the program starts it can overwrite its code or its data; in fact, you could manage quite well without indirect addressing in JBit (just patch the code on the fly).

On a real device a program would likely be stored in ROM (i.e. the code and data pages of a program could not be overwritten). The rest of the memory (RAM) might not be initialized. Furthermore, on some real devices the amount of RAM might be limited to a few bytes.

In JBit either the CPU or the IO chip are working, but never together. When the CPU writes to some particular locations of the IO chip, the CPU stops and the IO chip starts working. When the IO chip has finished, the CPU continues. This is why in JBit it is important to write 0 to FRMDRAW once in a while: to allow the IO chip to draw the display.

On a real device the CPU and the IO chip would work together. When something happens (e.g. 100ms have passed) the IO chip notifies the CPU sending an *IRQ* (Interrupt ReQuest). The CPU then stops the normal program and calls a special routine (the *IRQ handler*).

Some real devices have *watchdog timers* to protect against bugs and failures. The application must write to a location once in a while to signal that it is still working as expected; failure to do so will cause the device to be reset.

The goal will be to use the HAL instead of the IO chip. The HAL will provide a watchdog timer and will take control of the main program. The rest of the application will be written as an IRQ handler (a.k.a. *event-driven* or *asynchronous* programming). Finally, we will program as if the code and the data pages cannot be overwritten and we will not rely on the rest of the memory being initialized with 0s.

Listing

```
   * 3:0 *        * CODE *
 0 LDA #11        TEST          3:0
 2 STA 10         HANDLER      3:11
 4 LDA #3         HALSTART    4:136
 6 STA 11         HALINIT     4:153
 8 JMP 4:136      HALABORT    4:173
11 INC 2:40       HALSTEP     4:199
14 RTS            * DATA *
   * 4:136 *      FRMDRAW      2:18
136 JSR 4:153     CONVIDEO     2:40
139 JSR 4:199     CONROW1      2:50
142 DEC 19        REGIMAGE     8:46
144 BEQ 4:173     ERRORMSG     8:66
146 LDA #0        * ZERO PAGE *
148 STA 2:18      IRQ-LO         10
151 BEQ 4:139     IRQ-HI         11
153 LDX #20       RAM-0          18
155 LDA 8:45,X    WDTMR          19
158 STA 11,X      ROM-VERS       30
160 DEX           ROM-FREQ       31
161 BNE 4:155     SHADOW         32
163 LDA #0        * CONSTANTS *
165 LDX #5        ASC-SPC        32
167 STA 31,X      VIDSIZE        40
169 DEX
170 BNE 4:167
172 RTS
173 LDX #40
175 LDA #32
177 STA 2:39,X
180 DEX
181 BNE 4:177
183 LDA 8:66,X
186 BEQ 4:194
188 STA 2:50,X
191 INX
192 BNE 4:183
194 STA 2:18
197 BEQ 4:194
199 JMP (0:10)
```

Type In

```
C 3:0
000: 169 011 133 010 169 003 133 011
008: 076 136 004 238 040 002 096 000

C 4:136
136: 032 153 004 032 199 004 198 019
144: 240 027 169 000 141 018 002 240
152: 242 162 020 189 045 008 149 011
160: 202 208 248 169 000 162 005 149
168: 031 202 208 251 096 162 040 169
176: 032 157 039 002 202 208 250 189
184: 066 008 240 006 157 050 002 232
192: 208 245 141 018 002 240 251 108
200: 010 000 000 000 000 000 000 000

D 8:48
048: 000 000 000 000 000 100 252 096
056: 218 242 102 182 190 224 254 246
064: 001 010 073 078 084 046 032 069
072: 082 082 079 082 000 000 000 000
```

Comment

The HAL has 22 registers ranging from address 10 to address 31 on page 0. IRQ-LO and IRQ-HI contain the address of the IRQ handler. RAM-0 is a single cell of RAM initialized with 0. WDTMR is a counter initialized to 100 and decremented by 1 every time the IRQ handler is called. ROM-VERS contains the version of the HAL. ROM-FREQ contains how many times per seconds the IRQ handler is called (e.g. 10 Hz). The other registers will be described on the next sheet.

Jumping to HALSTART starts the HAL. The HAL first calls HALINIT. HALINIT initializes the registers (except IRQ-LO and IRQ-HI) using a table (REGIMAGE) and clears a buffer (SHADOW, described on the next sheet).

After HALINIT returns, the main loop begins. The IRQ handler is invoked using indirect addressing (HALSTEP) and WDTMR is decremented. If WDTMR reaches 0 the HAL shows an error message and stops (HALABORT) otherwise the display is updated (thus waiting 100ms) and another iteration can begin.

The test code sets the IRQ handler (HANDLER) and starts the HAL. The test handler updates a character on the display. Since it fails to reset WDTMR to 100, after a while the HAL will stop.

Note that the handler is supposed to be short and quick. Programming in this way can be tricky; you may have to split the work to do in stages and keep track of the progress (RAM-0 can be very handy to recognize the first stage). Try to write a simple program in this way (e.g. *loop2*).

JBit E1 (5)

Branching

Here is a quick recap about branching. Branch operations add an offset to the PC if a status flag has a particular value.

	Set	Clear
Zero (Z)	BEQ	BNE
Negative (N)	BMI	BPL
Carry (C)	BCS	BCC
Overflow (V)	BVS	BVC

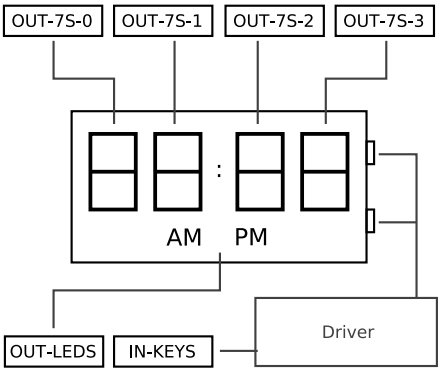
Zero. Modified by most operations: Load, Increment, Decrement, Addition, Subtraction, OR, AND, XOR, Comparison and Shifting. Set when the result is 0, cleared otherwise.
Negative. Modified by most operations, just like the Zero flag. Set or cleared depending on bit 7 of the result. In other words, set when the result is greater than or equal to 128, reset otherwise. 255, 254, ...128 are considered to be negative: -1, -2, ...-128.
Carry. Modified by Addition, Subtraction, Comparison and Shifting. Carry is used as an "Inverse Borrow" in subtractions (i.e. must be set before the operation and is reset if a borrow has occurred).
Overflow. Modified by Addition, Subtraction and BIT. During additions and subtractions the overflow is used to check if the result is outsize the range, please refer to other sources for more information. The BIT operation sets or clears the Overflow flag depending on bit 6 of a memory location.
Offset. To compute the offset, start with the operation after the branch (0) and count the number of bytes to jump forward (1, 2, 3...) or backward (255, 254, 253...). Note that, unlike JSR and JMP, the address to jump to is encoded as the number of bytes to skip, making the code easy to relocate. For short jumps it is common to use a branch using a flag with a known value (or CLV+BVC).

In the following example, BEQ 0 has no effect and BEQ 254 enters an infinite loop if 2:24 contains 0 (the Zero flag is set).

```
251 (-5) LDA 2:24
254 (-2) BEQ ?
  0 ( 0) STX 2:24
  3 ( 3) CLC
```

Hardware Abstraction Layer (2)

The last part of the HAL is about the handling of the Input/Output. First of all a disclaimer: I am not a programmer of embedded systems. The following considerations are, to the best of my knowledge, reasonable.



On a real device two buttons are likely to be mapped to two bits (e.g. 5 and 6) of an input register. 1 could mean pressed and 0 released (but it could well be the opposite).

Unfortunately, JBit does not provide a low level access to the keypad, so we opt for a simple mapping of the 8-byte KEYBUF to a single register (IN-KEYS). In other words, we assume that a driver is monitoring the buttons looking for bit transitions (e.g. if bit 5 changed from 0 to 1 since the last time it has been checked, MODE has been pressed) and posting events (0: nothing has happened, 1: SET has been pressed, 2: MODE has been pressed) to IN-KEYS accordingly.

To keep the number of output registers down, on a real device some sort of multiplexing is likely to be in place. One register would act as a selector and another would act as the bitmask for the selected led group. The CPU would have to keep refreshing every led group in turn.

Again, we opt for a simpler design. The HAL will provide 5 output registers: 4 bitmasks for the 4 digits and 1 bitmask for the other leds. 10 ROM registers will provide the standard bitmasks for the digits.

It is my understanding that is increasingly common to have intelligent I/O chips performing complex functions, so the approach we take here might not be totally unrealistic.

Listing

```
    * 3:0 *         * CODE *
  0 LDA #11         TEST          3:0
  2 STA 10          HANDLER      3:11
  4 LDA #3          VLDSDRAW     4:36
  6 STA 11          SSEGDRAW     4:63
  8 JMP 4:136       HALSTART    4:136
 11 LDA #100        HALSTEP     4:199
 13 STA 19          * DATA *
 15 LDX 17          KEYBUF        2:24
 17 BEQ 3:23        DSP-DGTS      8:77
 19 LDA 20,X        * ZERO PAGE *
 21 STA 13          IRQ-LO          10
 23 RTS             IRQ-HI          11
    * 4:199 *       OUT-7S-0        12
199 LDA 2:24        OUT-7S-1        13
202 BEQ 4:227       OUT-7S-2        14
204 CMP #51         OUT-7S-3        15
206 BNE 4:212       OUT-LEDS        16
208 LDA #1          IN-KEYS         17
210 BNE 4:222       RAM-0           18
212 CMP #57         WDTMR           19
214 BNE 4:220       ROM-7S-0        20
216 LDA #2          ROM-7S-1        21
218 BNE 4:222       ROM-7S-2        22
220 LDA #0          ROM-7S-3        23
222 LDX #1          ROM-7S-4        24
224 STX 2:24        ROM-7S-5        25
227 STA 17          ROM-7S-6        26
229 JSR 5:16        ROM-7S-7        27
232 LDA #4          ROM-7S-8        28
234 STA 50          ROM-7S-9        29
236 LDX 50          ROM-VERS        30
238 LDA 12,X        ROM-FREQ        31
240 CMP 32,X        SHADOW          32
242 BEQ 5:11        CURR-REG        50
244 STA 32,X        CURR-VAL        51
246 STA 51          * CONSTANTS *
248 CPX #4          ASC-3           51
250 BNE 5:2         ASC-9           57
252 JSR 4:36        KEY-MODE         2
255 JMP 5:11        KEY-SET          1
  2 LDA 8:77,X      MSK-AM           1
  5 TAX             MSK-DOTS         4
  6 LDA 51          MSK-PM           2
  8 JSR 4:63        REG-LEDS         4
 11 DEC 50
 13 BPL 4:236
 15 RTS
 16 JMP (0:10)
```

Type In

```
C 3:0
000: 169 011 133 010 169 003 133 011
008: 076 136 004 169 100 133 019 166
016: 017 240 004 181 020 133 013 096

C 4:192
192: 208 245 141 018 002 240 251 173
200: 024 002 240 023 201 051 208 004
208: 169 001 208 010 201 057 208 004
216: 169 002 208 002 169 000 162 001
224: 142 024 002 133 017 032 016 005
232: 169 004 133 050 166 050 181 012
240: 213 032 240 023 149 032 133 051
248: 224 004 208 006 032 036 004 076
000: 011 005 189 077 008 170 165 051
008: 032 063 004 198 050 016 221 096
016: 108 010 000 000 000 000 000 000

D 8:72
072: 082 082 079 082 000 000 002 005
080: 007 000 000 000 000 000 000 000
```

Comment

HALSTEP is rewritten to handle Input/Output.

Before calling the IRQ Handler, KEYBUF is mapped to IN-KEYS. This code is a bit unstructured so keeping in mind what should be done is helpful. If a KeyPress is available, 1 should be written to KEYBUF to consume it. Keys 3 and 9 should be mapped to 1 and 2. Any other key should be mapped to 0.

After calling the IRQ Handler, the display is updated. A buffer (SHADOW, 5 bytes) is used to store the old values of the OUT registers. Every OUT register is checked in turn. If it differs from its shadow, the shadow is updated and the relevant subroutine (VLDSDRAW or SSEGDRAW) is called. Since these subroutines might change the CPU registers, some support cells (CURR-REG and CURR-VAL) are used.

The test handler checks if a key if pressed and updates the second 7 Segment Display accordingly. It also resets the watchdog timer.

JBit E1 (6)

???

To be written...

The Clock

...

The model stores the current time of the day. Choosing how to represent the model can be very tricky. Execution speed and easy of programming can be influenced heavily by this choice. It is not uncommon to gain in one section of the program and lose in another. At the end I have chosen the following scheme:

HH ranges from 0 to 23. Instead of using a single MM field ranging from 0 to 59, two fields are used: M1 ranging from 0 to 5 and M0 ranging from 0 to 9. Same for S1 and S0. TCKS ranges from 0 to ROM-FREQ - 1 (e.g. 0 to 49 for a 50Hz HAL).

Here is how 9:53:26.4PM would be represented on a 50Hz HAL:

TCKS	S0	S1	M0	M1	HH
20	6	2	3	5	21

The valid values for the STATUS field are: START (i.e. the clock is not yet initialized), SETHH / SETM1 / SETM0 (i.e. the clock is being set) and HHMM / SS (i.e. the clock is ticking).

STATUS will be stored in RAM0 and 0 will be used for START. From now on, RAM0 will be referenced as STATUS.

...

Listing

```
   * 3:0 *        * CODE *
 0 LDA #11        TEST          3:0
 2 STA 10         HANDLER      3:11
 4 LDA #3         HALSTART    4:136
 6 STA 11         MDL-INIT     5:19
 8 JMP 4:136      MDL-GO       5:28
11 LDA 18         MDL-TICK     5:37
13 BNE 3:24       * ZERO PAGE *
15 JSR 5:19       MDL-TCKS        1
18 JSR 5:28       MDL-S0          2
21 INC 18         MDL-S1          3
23 RTS            MDL-M0          4
24 JSR 5:37       MDL-M1          5
27 LDX 2          MDL-HH          6
29 LDA 20,X       IRQ-LO         10
31 STA 15         IRQ-HI         11
33 RTS            OUT-7S-3       15
   * 5:19 *       STATUS         18
19 LDA #0         ROM-7S-0       20
21 STA 4          ROM-FREQ       31
23 STA 5
25 STA 6
27 RTS
28 LDA #0
30 STA 1
32 STA 2
34 STA 3
36 RTS
37 LDX #0
39 INC 1
41 LDA 1
43 CMP 31
45 BNE 5:99
47 STX 1
49 INC 2
51 LDA 2
53 CMP #10
55 BNE 5:99
57 STX 2
59 INC 3
61 LDA 3
63 CMP #6
65 BNE 5:99
67 STX 3
69 INC 4
71 LDA 4
73 CMP #10
75 BNE 5:99
77 STX 4
79 INC 5
81 LDA 5
83 CMP #6
85 BNE 5:99
87 STX 5
89 INC 6
91 LDA 6
93 CMP #24
95 BNE 5:99
97 STX 6
99 RTS
```

Type In

```
C 3:0
000: 169 011 133 010 169 003 133 011
008: 076 136 004 165 018 208 009 032
016: 019 005 032 028 005 230 018 096
024: 032 037 005 166 002 181 020 133
032: 015 096 000 000 000 000 000 000
```

```
C 5:16
016: 108 010 000 169 000 133 004 133
024: 005 133 006 096 169 000 133 001
032: 133 002 133 003 096 162 000 230
040: 001 165 001 197 031 208 052 134
048: 001 230 002 165 002 201 010 208
056: 042 134 002 230 003 165 003 201
064: 006 208 032 134 003 230 004 165
072: 004 201 010 208 022 134 004 230
080: 005 165 005 201 006 208 012 134
088: 005 230 006 165 006 201 024 208
096: 002 134 006 096 000 000 000 000
```

Comment

The subroutines MDL-INIT and MDL-GO initialize the model. The subroutine MDL-TICK increments MDL-TCKS. If MDL-TCKS reaches the frequency of the IRQ (ROM-FREQ), this means that a whole second has elapsed; MDL-TCKS is cleared and MDL-S0 is incremented. If necessary, the process (reset and increment) is applied to the other fields (MDL-S0, MDL-S1, MDL-M0, MDL-M1, MDL-HH) using the appropriate thresholds (10, 6, 10, 6, 24).

The first time the test handler is invoked, the model is initialized and STATUS is updated to 1. During normal operation, the model is updated (MDL-TICK) and the MDL-S0 field is then used to drive the last digit of the display (OUT-7S-3).

JBit E1 (7)

Listing

```
   * 3:0 *
 0 LDA #11
 2 STA 10
 4 LDA #3
 6 STA 11
 8 JMP 4:136
11 LDA #100
13 STA 19
15 LDA 18
17 BEQ 3:47
19 LDA 17
21 CMP #1
23 BEQ 3:30
25 CMP #2
27 BEQ 3:41
29 RTS
30 INC 18
32 LDA 18
34 CMP #4
36 BEQ 3:50
38 JMP 5:121
41 JSR 5:100
44 JMP 5:121
47 JSR 5:19
50 LDA #1
52 STA 18
54 JMP 5:121
   * 5:100 *
100 LDY 18
102 DEY
103 LDA 8:81,Y
106 TAX
107 INC 0,X
109 LDA 0,X
111 CMP 8:84,Y
114 BNE 5:120
116 LDA #0
118 STA 0,X
120 RTS
121 LDX #0
123 LDA 18
125 CMP #1
127 BEQ 5:146
129 CMP #2
131 BEQ 5:153
133 CMP #3
135 BEQ 5:158
137 CMP #4
139 BEQ 5:161
141 CMP #5
143 BEQ 5:171
145 RTS
146 STX 14
148 STX 15
150 JMP 5:190
153 STX 15
155 JMP 6:5
158 JMP 6:12
161 JSR 5:190
164 JSR 6:5
167 JSR 6:12
170 RTS
171 STX 12
173 STX 13
175 STX 16
177 LDX 3
179 LDA 20,X
181 STA 14
183 LDX 2
185 LDA 20,X
187 STA 15
189 RTS
190 STX 12
192 LDY #4
194 LDA 18
196 CMP #4
198 BNE 5:209
200 LDA 31
202 LSR
203 CMP 1
205 BPL 5:209
207 LDY #0
209 TYA
210 LDY 6
212 CPY #12
214 BMI 5:228
216 ORA #2
218 STA 16
220 LDA 6
222 SEC
223 SBC #12
225 JMP 5:234
228 ORA #1
230 STA 16
232 LDA 6
```

```
   * CODE *
TEST          3:0
HANDLER       3:11
HALSTART      4:136
MDL-INIT      5:19
MDL-EDIT      5:100
UPD-VIEW      5:121
UV-SETHH      5:146
UV-SETM1      5:153
UV-SETM0      5:158
UV-HHMM       5:161
UV-SS         5:171
UV-HHLDS      5:190
UV-MIN1        6:5
UV-MIN0       6:12
   * DATA *
TAB-EFLD      8:81
TAB-EMAX      8:84
   * ZERO PAGE *
MDL-TCKS         1
MDL-S0           2
MDL-S1           3
MDL-M0           4
MDL-M1           5
MDL-HH           6
IRQ-LO          10
IRQ-HI          11
OUT-7S-0        12
OUT-7S-1        13
OUT-7S-2        14
OUT-7S-3        15
OUT-LEDS        16
IN-KEYS         17
STATUS          18
WDTMR           19
ROM-7S-0        20
ROM-7S-1        21
ROM-7S-2        22
ROM-FREQ        31
   * CONSTANTS *
KEY-MODE         2
KEY-SET          1
MSK-AM           1
MSK-DOTS         4
MSK-PM           2
ST-HHMM          4
ST-SETHH         1
ST-SETM0         3
ST-SETM1         2
ST-SS            5
ST-START         0
```



```
234 BEQ 5:246
236 CMP #10
238 BPL 5:248
240 TAX
241 LDA 20,X
243 STA 13
245 RTS
246 LDA #12
248 SEC
249 SBC #10
251 TAX
252 LDA 20,X
254 STA 13
  0 LDA 21
  2 STA 12
  4 RTS
  5 LDX 5
  7 LDA 20,X
  9 STA 14
 11 RTS
 12 LDX 4
 14 LDA 20,X
 16 STA 15
 18 RTS
```

Type In

```
C 3:0
000: 169 011 133 010 169 003 133 011
008: 076 136 004 169 100 133 019 165
016: 018 240 028 165 017 201 001 240
024: 005 201 002 240 012 096 230 018
032: 165 018 201 004 240 012 076 121
040: 005 032 100 005 076 121 005 032
048: 019 005 169 001 133 018 076 121
056: 005 000 000 000 000 000 000 000

C 5:96
096: 002 134 006 096 164 018 136 185
104: 081 008 170 246 000 181 000 217
112: 084 008 208 004 169 000 149 000
120: 096 162 000 165 018 201 001 240
128: 017 201 002 240 020 201 003 240
136: 021 201 004 240 020 201 005 240
144: 026 096 134 014 134 015 076 190
152: 005 134 015 076 005 006 076 012
160: 006 032 190 005 032 005 006 032
168: 012 006 096 134 012 134 013 134
176: 016 166 003 181 020 133 014 166
184: 002 181 020 133 015 096 134 012
192: 160 004 165 018 201 004 208 009
200: 165 031 074 197 001 016 002 160
208: 000 152 164 006 192 012 048 012
216: 009 002 133 016 165 006 056 233
224: 012 076 234 005 009 001 133 016
232: 165 006 240 010 201 010 016 008
240: 170 181 020 133 013 096 169 012
248: 056 233 010 170 181 020 133 013
000: 165 021 133 012 096 166 005 181
008: 020 133 014 096 166 004 181 020
016: 133 015 096 000 000 000 000 000

D 8:80
080: 007 006 005 004 024 006 010 000
```

Comment

The subroutine MDL-EDIT increments a field and if the field reaches a threshold (e.g. 24) it resets it. The fields to edit and the relevant thresholds are stored in two tables (TAB-EFLD and TAB-EMAX, indexed by STATUS; only ST-SETxx are valid).

The subroutine UPD-VIEW is long, but most of it is simple. First STATUS is checked and the control is transfered to the relevant code. See if you can understand all the cases; starting with UV-SS (the easiest case, not further decomposed) and ending with UV-HHLDS (the hardest section of code). Mostly, it is a matter of using some fields of the model to drive the output registers accordingly.

UV-HHLDS drives the first two digits and the leds according to MDL-HH, MDL-TCKS and STATUS. First the status of the DOTS led is is computed; it is turned off only if STATUS is ST-HHMM and MDL-TCKS is less than half ROM-FREQ, causing blinking. Then the MDL-HH field is mapped from the 0-23 range to the 0-11 range, setting the AM/PM led accordingly. Finally the HH digits are computed. For values less than 10, the second digit is ready available (the first digit was previously cleared). For other values, the digit is computed by subtracting 10. Note that 0 is mapped to 12.

The test handler reacts to the input keys. When KEY-SET is pressed, the STATUS cycles through ST-SETHH (the initial value), ST-SETM1 and ST-SETM0. When KEY-MODE is pressed, MDL-EDIT is called. At the end of any interrupt, UPD-VIEW is called.

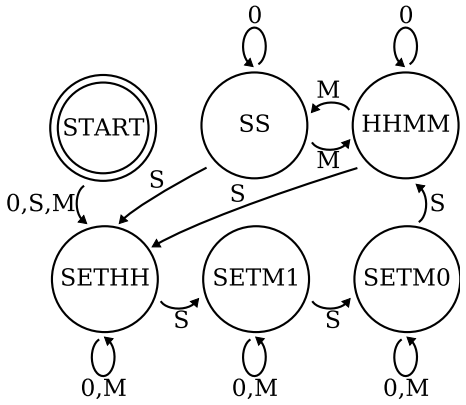
JBit E1 (8)

Finite State Machines

You might be tempted to start writing the controller using lots of comparisons and branchings. This is an intuitive approach, but often results in code that is difficult to read. It is also easy to miss a case.

A more ordered approach is to use Finite State Machines. FSMs come from parsing technology, but they are used in other contexts as well.

You start with a *State Transition Diagram* that shows how the status changes when a given input value is received:



Every state must be included and every input value must be handled for each state. It is far easier to make sure of that with a diagram than with several pages of code. The initial state is usually marked.

Once you have checked the diagram, you can rewrite it as a *State Transition Table*:

	0	**SET**	**MODE**
START	SETHH	SETHH	SETHH
SETHH	SETHH	SETM1	SETHH
SETM1	SETM1	SETM0	SETM1
SETM0	SETM0	HHMM	SETM0
HHMM	HHMM	SETHH	SS
SS	SS	SETHH	HHMM

After a bit of practice, you should even be able to read and write a State Transition Table without using a State Transition Diagram.

A similar approach can also be used to describe how the model is modified:

	0	**SET**	**MODE**
START	INIT	INIT	INIT
SETHH	-	-	EDIT
SETM1	-	-	EDIT
SETM0	-	GO	EDIT
HHMM	TICK	-	TICK
SS	TICK	-	TICK

Conclusion

As the name of the series implies, there might be other serieses in the future, but in my opinion this will always be *the* JBit tutorial. Once you master the content of this series and you are able to write a similar program on your own, you might even want to consider if JBit has already given you the best it has to offer. Here are some suggestions about where to go from here:

6502. Fill the blanks left in this series (e.g. binary numbers, stack, overflow, interrupts, etc...).

MacroAssembler. Use a MacroAssembler to write a program on your PC and pack it into a MIDlet. I suggest cc65. Understanding the cc65 toolchain (configuration file, assembler and linker) might seem daunting at first, but eventually it will make sense and will easy the transition to the GCC toolchain. cc65 will also easy the transition to C (see below).

IO Chip. Write a simple game. Unfortunately, at the time of this writing the documentation of the IO chip is lacking and you are pretty much left on your own. Looking at the demo programs might help.

Assembly and Computer Architecture. Learn a modern 32/64 bit Assembly. There is usually no need to be able to write programs with it, but some concepts are good to know (e.g. branch prediction, cache, memory models, etc...).

C Language. While C is now a niche language, it is still very useful. The syntax of the language (the hardest part for a beginner) is a sort of lingua franca and is the basis of some modern languages (e.g. Java, C# and JavaScript). The rest of the language is not too hard if you already know Assembly; C is sometimes called "Portable Assembly" for a reason.

Software Engineering. Programming Techniques are only hinted here. Read a good book on Software Engineering for a better introduction.

Listing

```
   * 3:0 *        * CODE *
 0 JMP 6:19       BOOT          3:0
   * 6:19 *       HALSTART    4:136
19 LDA #31        MDL-INIT     5:19
21 STA 10         MDL-GO       5:28
23 LDA #6         MDL-TICK     5:37
25 STA 11         MDL-EDIT    5:100
27 JMP 4:136      UPD-VIEW    5:121
30 RTS            MAIN         6:19
31 LDA #100       NO-FUNC      6:30
33 STA 19         HANDLER      6:31
35 CLC            * DATA *
36 LDA 18         TAB-F-LO     8:87
38 ADC 18         TAB-F-HI    8:105
40 ADC 18         TAB-STTR    8:123
42 ADC 17         * ZERO PAGE *
44 STA 52         IRQ-LO         10
46 LDX 52         IRQ-HI         11
48 LDA 8:87,X     IN-KEYS        17
51 STA 53         STATUS         18
53 LDA 8:105,X    WDTMR          19
56 STA 54         OFFSET         52
58 JSR 6:71       TMP-F-LO       53
61 LDX 52         TMP-F-HI       54
63 LDA 8:123,X    * CONSTANTS *
66 STA 18         ST-HHMM         4
68 JMP 5:121      ST-SETHH        1
71 JMP (0:53)     ST-SETM0        3
                  ST-SETM1        2
                  ST-SS           5
```

Type In

```
C 3:0
000: 076 019 006 000 000 000 000 000

C 6:16
016: 133 015 096 169 031 133 010 169
024: 006 133 011 076 136 004 096 169
032: 100 133 019 024 165 018 101 018
040: 101 018 101 017 133 052 166 052
048: 189 087 008 133 053 189 105 008
056: 133 054 032 071 006 166 052 189
064: 123 008 133 018 076 121 005 108
072: 053 000 000 000 000 000 000 000

D 8:80
080: 007 006 005 004 024 006 010 019
088: 019 019 030 030 100 030 030 100
096: 030 028 100 037 030 037 037 030
104: 037 005 005 005 006 006 005 006
112: 006 005 006 005 005 005 006 005
120: 005 006 005 001 001 001 001 002
128: 001 002 003 002 003 004 003 004
136: 001 005 005 001 004 000 000 000
```

Comment

The boot code jumps to the main code. The main code registers the IRQ Handler and starts the HAL. The handler resets the watchdog timer, computes the offset of the lookup tables, updates the model, updates the status and updates the view.

The logic of the controller is encoded in three tables: TAB-F-LO and TAB-F-HI for the model transition and TAB-STTR for the state transition. The tables are stored in row major order (i.e. first the first row from left to right, then the second row and so on).

The offset is computed using the following formula: $3 \times status + input$. Since the 6502 has no multiplication, the equivalent: $status + status + status + input$ is computed. If the formula was more complex (e.g. 7 inputs), a lookup table could have been used to compute the multiplication.

