Micro-KIM Tutorial

Aart J.C. Bik
http://www.aartbik.com/

1 Getting Started

Perhaps reminiscing the past is a sign of getting older, but I cannot help but look back fondly at the times I learned programming machine code on the Commodore 64 in the eighties. Therefore, it is probably no surprise I still occasionally enjoy programming 6502 on the **Micro-KIM**, which is a modern replica of the seventies KIM1 microcomputer, made available by the well-known retro computer kits provider Briel Computers [**?**].

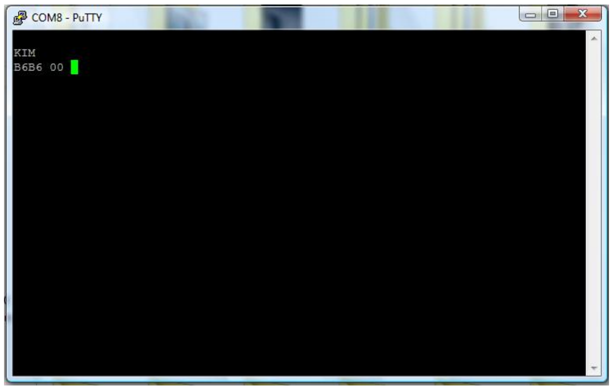
In fact, I am having so much fun with this board, I decided to write a series of tutorials on operating and programming the Micro-KIM. In this series, I assume you have already some experience with the Micro-KIM and 6502 machine code, and have read the basic documentation that is shipped with the kit. Other than that, I hope to give additional information on various topics, such as developing assembly programs, programming the display, using the RS232 port or keypad, setting up timer-based interrupts, using a cross-assembler to generate programs in paper tape format, and uploading these to the kit.

Note that the original KIM1 featured a 6502 microprocessor, 1K of static RAM, two 6530 RRIOT IC's, and a 6 character hexadecimal LED diplay. Even though the Micro-KIM is a surprisingly accurate clone, it features a single 6532 RIOT, 2K EPROM for the monitor program, and 5K RAM. Please keep these differences in mind while reading the tutorial, since not all examples that work on the Micro-KIM will also work on the original KIM1.

Let's get started by uploading a simple demo to the Micro-KIM. In following tutorials, I plan to delve into the details of the demo itself. For now, I simply give the demo in paper tape format.

```
;180200A97F8D4117A93F8D4317A90A85E7A200A0098C4217BD4802094C
;1802188D4017203F02E8C8C8E007D0EDC6E7D0E5A200BD48029D470C82
;18023002E8E007D0F5AD47028D4E024C0A0298A0C888D0FDA860080B70
;0702480840014008080800F2
;0000040004
```

First, connect the Micro-KIM through a serial cable with your computer, make sure the jumper JP2 is in the ON position to enable RS232 input in the monitor program, and switch the kit on. Next, start a terminal program on your computer, such as HyperTerminal or PuTTY, which I assume you have set up already as described in the basic documentation. Last, press the RS key on the kit and hit ENTER in the terminal program. If all goes well, the Micro-KIM greets you with a prompt that looks something like Figure 1.



```
KIM
B6B6 00
```

Figure 1: TTY Serial Interface: Micro-KIM Prompt

Then copy the paper tape format code above into the clipboard, press L in the terminal, and then paste from the clipboard. Again, if all goes well, eventually the Micro-KIM answers with the prompt again. Then press 0200 SPACE and G to start the program. Alternatively, you can remove the jumper, press 0200 on the keypad on the kit and GO to start this program. Both ways will work for this particular demo, which looks as shown in Figure 2.

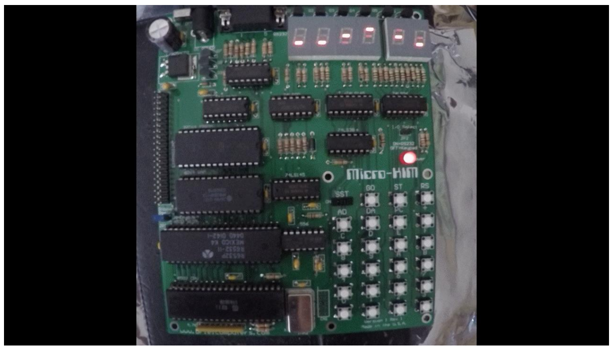


Figure 2: Wave Demo on the Micro-KIM

In the next tutorial, I will introduce cross-assemblers and start exploring how to program the display.

2 A First Assembly Program

At the lowest level, the 6502 executes numerical machine code. For example, the following bytes in hexadecimal format constitute a simple program that displays a single 8 on the LED display of the Micro-KIM.

```
a9 ff 8d 40 17 a9 09 8d 42 17 4c 0a 02
```

Let's enter this program into the memory of the Micro-KIM. Power on the kit with jumper JP2 off and press the RS key. Then enter 0200 to set the address and press DA to go into data mode. Next, enter the numbers above pressing the + key after each number pair (so, enter A9 + FF + etc.). Before running, I strongly recommend checking the values. Use AD to go back into address mode. Press 0200 again and use + repeatedly to check all entered values. Once satisfied, press 0200 and GO. If all goes well, you will see a very bright 8 as first digit on the LED display (in later tutorials I will explain why), as shown in Figure 3.

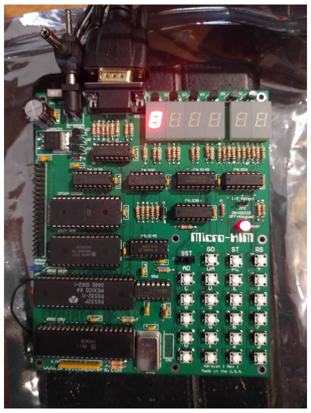


Figure 3: Displaying a Single Digit on the Micro-KIM

Obviously constructing and entering programs this way is tedious and error-prone. It is much easier to program in assembly language, where statements still closely relate to machine instructions, but where an assembler takes care of translating instructions and addressing modes into the numeric equivalent as well as resolving symbolic names and evaluating simple expressions into actual values. For the Micro-KIM, one typically wants to use a cross-assembler, i.e., an assembler that runs on a host computer, such as a desktop or laptop, but generates machine code for a different target computer, in this case the Micro-KIM.

While reliving the good old days of my Commodore 64, I implemented a cross-assembler that runs on Windows (win2c64), Linux (lin2c64), and MacOS (mac2c64) and generates machine code for a 65xx-based microcomputer [**?**]. This assembler supports several common output formats, including the paper tape format that can be directly uploaded to the Micro-KIM. Therefore, in this series, I will use win2c64 for assembly.

The assembly code for the program above looks as follows (note, if you are a bit rusty on the 6502, I recommend reading some online documentation on the instruction set first; more details on the assembler syntax and operation of win2c64 can be found in the online manual; details on the program itself will follow in later tutorials).

```
sad  .equ $1740
sbd  .equ $1742
     .org $0200         ; start program at $0200
     lda #$ff
     sta sad            ; set all bits of A data
     lda #9
     sta sbd            ; set value 9 in B data
loop jmp loop           ; loop forever to avoid
                        ; returning to monitor program
```

To invoke the assembler and generate paper tape format, save this code in a source file, kim.s, and then run the following from the command line.

```
win2c64 -P kim.s
```

This generates an output file kim.ptf in textual paper tape format (there is also a binary variant, but that one is less useful for uploading over TTY). The contents of this file are shown below. These can be copied-and-pasted to the Micro-KIM through the terminal, as explained in the first tutorial. Much more convenient then entering a long list of numbers!

```
;0D0200A9FF8D4017A9098D42174C0A02048B
;0000010001
```

The win2c64 binary can also be used as disassembler on the generated output as follows.

```
win2c64 -dP kim.ptf
```

Which shows the addresses, numeric encoding and instructions as follows (note how the assembler has resolved and removed all the symbolic information and comments from the original source file).

```
$0200 a9 ff     lda #$ff
$0202 8d 40 17  sta $1740
$0205 a9 09     lda #$09
$0207 8d 42 17  sta $1742
$020a 4c 0a 02  jmp $020a
```

Just looking at the encoding, you may recognize the numbers you entered manually at the start of this tutorial.

Now that you have become more familiar with the tools, you are ready for the next tutorial, where I will explore using routines from the monitor program to show more numbers on the LED display in your assembly program. After that, I will explore taking full control of the LED display!

3 The Monitor Program

A simplified memory map of the Micro-KIM is shown below. This tutorial explores the 2K EPROM, leaving a more detailed exploration of the free RAM and 6532 RIOT for later. Address space $1400 to $173f is unused in the standard Micro-KIM kit configuration.

```
+-----------+
| 2K EPROM  |$1fff
| monitor   |
| program   |$1800
+-----------+
| 6532 RIOT |$17ff
| I/O, timer|
| and RAM   |$1740
+-----------+
| optional  |$173f
| I/O, timer|
| and RAM   |$1400
+-----------+
|           |$13ff
|  5K RAM   |
|           |$0000
+-----------+
```

Addresses $1800 through $1fff are taken by the 2K EPROM, which is a read-only memory area that stores the 6530-003 and 6530-002 parts of the monitor program. You can, of course, inspect all individual bytes in the address mode on the Micro-KIM kit, but I recommend reading the assembly listing of the monitor program in the appendix of the Setup and User's Manual of Briel Computers. The compact coding style found in this monitor program is quite educational, but the monitor program also provides a rich set of subroutines that can be used in your own programs.

When you press the reset key RS on the kit (RST signal), the 6502 processor jumps to the address stored in $1ffc/$1ffd, which has the hard-coded value $1c22 in ROM. This is the entry labeled RST in the monitor program, i.e. the KIM entry via reset. Similarly, the non-maskable interrupt (NMI signal) and interrupt request (IRQ signal) jump to addresses stored in $1ffa/$1ffb and $1ffe/$1fff, respectively, with hard-codes values $1c1c and $1c1f in ROM. These are the entries labeled NMIT and IRQT in the monitor program, which contain indirect jump instructions to the vectors NMIV and IRQV stored in RAM at addresses $17fa/$17fb and $17fe/$17ff. Since these entries in RAM are undefined on power on, the User's Manual instructs you to fill these RAM locations with the value $1c00, so that the ST key or single-step feature (NMI) or BRK instruction (IRQ) jump into the entry labeled SAVE in the monitor program.

The JP2 jumper selects whether the monitor program should handle communication over the RS232 port (jumper on) or keypad/display on the kit (jumper off). Note that both the RS232 port and keypad/display always can be programmed to work regardless of this jumper state. But the jumper is connected with bit PA0 of data port A in the 6532 RIOT, which is tested in the monitor program to decide what action to perform next.

A very useful subroutine in the monitor program labeled SCANDS appears at address $1f1f. Even though these instructions are actually part of a larger subroutine that is used to show the addresses and data during normal operation of the kit, when calling this entry directly, the kit shows the three bytes stored consecutively in zero page addresses $f9, $fa, and $fb in hexadecimal format on the 6 digit LED display.

This routine makes writing a simple three-byte counter very easy, as shown below.

```
scands .equ $1f1f
       .org $0200
;
; Initialize a 3 byte counter to zero.
;
       lda #0
       sta $f9
       sta $fa
       sta $fb
;
; Display and increment the 3 byte counter in a loop.
; Displaying before each increment slows down counting
; quite a bit.
;
loop   jsr scands
       inc $f9
       bne loop
       inc $fa
       bne loop
       inc $fb
       jmp loop
```

Using the cross-assembler as shown in the previous tutorial yields the following paper tape output.

```
;180200A90085F985FA85FB201F1FE6F9D0F9E6FAD0F5E6FB4C08020F22
;0000010001
```

Uploading this to the kit and running looks as shown in Figure 4, counting up a 6 digit (three byte) hexadecimal number. Although the lower digit goes faster than the eye can see, calling the display subroutine before each increment substantially slows down counting. A follow up tutorial looks at using interrupts for display, making the actual computation, counting in this case, much faster.

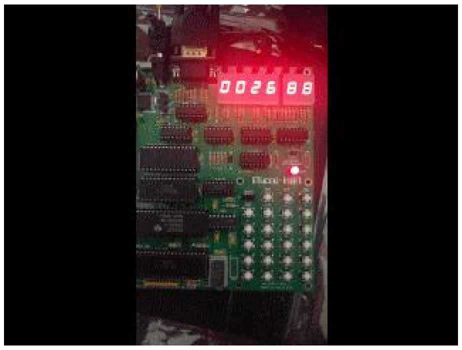


Figure 4: Slow Counter on the Micro-KIM

That's it for now. In the next tutorial, I am going to show how to use the 6532 RIOT to take full control of the LED display, show custom-made characters at any position, control brightness, and avoid flickering.

4 The LED Display

The schematic in Figure 5 illustrates what is fun about retro computing: the complete schematic of a microcomputer fits on a single page (a higher resolution PDF can be downloaded from the Briel Computers website [?]).

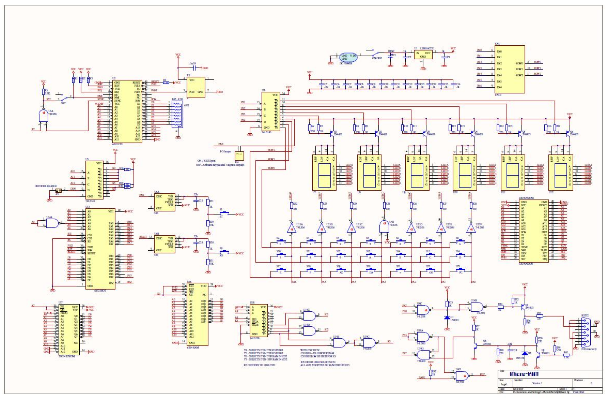


Figure 5: Micro-KIM Schematic. Courtesy Vince Briel - Briel Computers

The schematic shows that the 6 character LED display is controlled through some selection logic by the data ports of the 6532 RIOT. Because the 16 pins of the two 8-bit data ports A and B would not have sufficed to control all characters in the LED display simultaneously, instead a few bits of B select one character (value 9 selects the first, value 11 the second, etc.) while the lower 7 bits in A are used to control the 7 segments of that particular character (bit 0 controls the top segment, bit 1 upper right segment, etc.).

Note that with this scheme, it is possible to set one character and "go on with the program", as I showed in an earlier tutorial by displaying a very bright 8 in the first character, followed by simply looping the program (it could do something else instead). However, it is not possible to set all characters somehow, and "go on". Instead, the program has to loop over all characters and constantly refresh their contents. Due to the refreshing loop, displaying the full display appears a bit less bright than displaying a single character without such a loop.

Now let's write down some code to control the full LED display. First, some definitions provide symbolic names for the addresses of the data ports of the 6532 RIOT. The data registers contain the actual values, whereas the bits of the data direction registers define whether each pin is used for input (0) or output (1).

```
sad  = $1740   ; A data register
padd = $1741   ; A data direction register
sbd  = $1742   ; B data register
pbdd = $1743   ; B data direction register
```

Next some initialization code sets the 6532 RIOT data direction registers for output on the needed pins.

```
.org $0200
             lda #$7f
             sta padd
             lda #$3f
             sta pbdd
```

Then, the refreshing loop looks as follows. Here, register x iterates from 0 to 5 to load the proper value for the 7 segments of each character from a data array (with values that define the string "aart b"). Register y iterates from 9 to 19 with increment 2 to select each subsequent character on the LED display through data register B. Note that before changing data register B, the program clears data register A to ensure the old contents do not accidentally "flicker" very briefly in the next character. Furthermore, the program has a short delay when each next character is shown to ensure that character "glows up" a bit before moving on.

```
display_loop ldx #0
             ldy #9
char_loop    lda #0
             sta sad         ; no flicker
             sty sbd
             lda data, x
             sta sad
             txa
             ldx #4
char_delay   dex
             bne char_delay  ; glow up character
             tax
             inx
             iny
             iny
             cpx #6
             bne char_loop

             jmp display_loop ; keep refreshing

data .byte   $f7 $f7 $d0 $f8 $00 $fc
```

Assembling, uploading, and running this program as shown in earlier tutorials shows the output shown in Figure 6 on the LED display. Of course, feel free to change the values in the data array to your own custom-made characters.

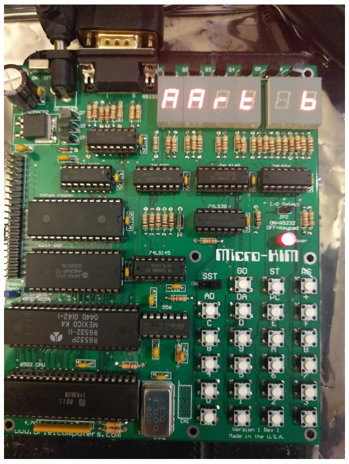


Figure 6: Taking Full Control of the LED Display on the Micro-KIM

That's it for this tutorial. Next tutorials will talk more on controlling the brightness of the LED display, scrolling text, moving graphics, and using interrupts to implement the refreshing loop.

5 Brightness of the LED Display

A demo is a program that shows off the abilities of a computer or programmer, sometimes even beyond the limits of an original architectural design. For example, a well-known demo theme on the Commodore 64 consists of rendering sprites in the border, i.e. outside the area originally destined for rendering sprites. This tutorial presents demos that use the LED display beyond its (probable) original purpose: adjusting the brightness of characters or even segments.

As shown in the previous tutorial, a refreshing loop is necessary to show all 6 characters on the LED display. Here, the refreshing rate directly defines the brightness of these characters. Simply looping around yields maximum brightness, while lowering the refresh rate dims the screen. This idea can also be used to adjust the brightness of parts of the LED screen (characters or even individual segments within the characters).

To illustrate this effect, let's modify the program of the previous tutorial. To focus on the brightness difference, only 'A's are shown. Also, rather than directly looping around, a few new instructions are added right before the jump back.

```
             lda #0
             sta sad
             lda #11
             sta sbd
             lda #$f7  ; change to $40 for segment
             sta sad
             ldx #200
bright_delay dex
             bne bright_delay

             jmp display_loop
```

This new code glows up the second character for a while before looping around refreshing all other characters. The result of this change is that the second A on the LED display looks a lot brighter, as illustrated in Figure 7 (the effect looks a bit better on the actual kit than in this picture).

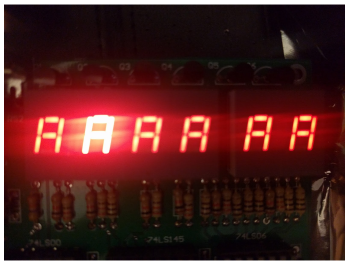


Figure 7: Single Bright Character (second A) on the Micro-KIM

The same idea can be used to change the brightness of individual segments within characters. Changing the value $f7 into $40 as indicated in the comment above makes the center segment of the second A brighter than the rest of the display, as shown in Figure 8 (again, the effect looks a bit better on the actual kit).

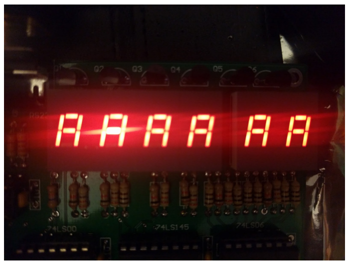


Figure 8: Single Bright Segment (center segment in second A) on the Micro-KIM

Finally, the idea gives rise to a wide variety of demos that change the brightness of the LED display somehow, such as fading the display in and out, sliding a glow effect over characters or segments, or giving emphasis to parts of the display. Give it a try and post your demo here as a comment. For starters, you can find an animated GIF of a sliding glow demo I wrote on the website shown in the contact section.

6 The Memory Map

Let's revisit the Micro-KIM memory map, introduced in the third tutorial.

```
+-----------+
| 2K EPROM  |$1fff
| monitor   |
| program   |$1800
+-----------+
| 6532 RIOT |$17ff
| I/O, timer|
| and RAM   |$1740
+-----------+
| optional  |$173f
| I/O, timer|
| and RAM   |$1400
+-----------+
|           |$13ff
|  5K RAM   |
|           |$0000
+-----------+
```

Since the default kit (without any expansion) only uses the lower address bits to access 8K, memory repeats itself every 8K. You can verify this by storing and inspecting values in, for instance, addresses $0000 and $2000. Any value stored in one address will show up in the other. Although an interesting factoid, there is no reason to let Micro-KIM programs address anything outside the range $0000-$1fff.

Addresses $0000-$13ff contain 5K free RAM (another interesting factoid: the Micro-KIM actually wastes 3K of its 8K RAM chip to keep compatibility with the original KIM-1). This memory region can be used to store data and code. To verify this, while running the monitor program, use the keyboard to store and inspect values. For example, type AD followed by 0200, DA, and FF and you will see the value at this address change. Type + a few times to change the address. Then, if you later revisit this address with AD 0200, you will still see the value FF stored at this address.

The first 256 bytes of RAM with addresses $0000-$00ff are called the zero page. The 6502 microprocessor supports zero-page addressing to access this part of RAM with more compact and more efficient instructions. For example, both the following two instructions load the contents of address $0000 into register X, but the first uses just 2 bytes rather than 3 bytes for the instruction encoding and executes in 3 rather than 4 cycles.

```
a6 00     ldx $00
ae 00 00  ldx $0000
```

This page is typically used by programs to store frequently accessed bookkeeping variables. In fact, the monitor program uses a few bytes stored at addresses $00ef-$00ff for various functions. The next page of RAM with addresses $0100-$01ff is used as stack by the 6502 microprocessor and should be left alone. From address $0200 all the way up to $13ff can be used freely though, which is why most programs start with the following command.

```
.org $0200
```

Addresses $1400-$173f are reserved for an optional second RIOT chip, but are otherwise unused in the default kit. You can verify this by trying to store anything in this range. Type AD, 1400, DA, followed by any value. No matter how hard you try, the data field in the LED display remains 00. Addresses $1740-$17ff are used by the 6532 RIOT chip. The functions of this peripheral chip are made available as regular memory addresses, a technique referred to as **memory mapping**. The 6532 provides 128 bytes of free RAM, two parallel I/O data ports, and programmable timers with interrupt capability. Details of this chip are given in later tutorials. For now, type AD 1744 while running the monitor program to get a surprise: the data field on the LED does not show one value, but cycles through many, as shown in Figure 9.

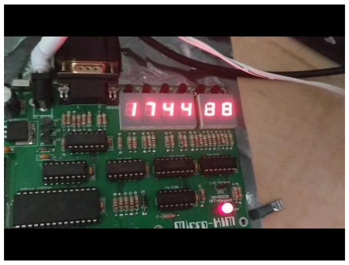


Figure 9: Memory Mapped Address in the Micro-KIM

Lastly, addresses $1800-$1fff are taken by the 2K EPROM that contains the monitor program, discussed earlier. Although you can use the monitor program to inspect its own values in this memory region, you can obviously not change the values by typing DA and a value. For example, after typing AD and 1FFF, the data field shows the value 1C corresponding to the second byte of IRQT, as can be seen in the listing of the monitor program in the appendix of the Setup and User's Manual of Briel Computers. However, typing DA followed by any value has no effect, since this part addresses immutable ROM.

7 Coming Soon!

The tutorial is still in progress. More content to follow soon!

8 Contact

If you enjoyed this tutorial or have suggestions for future topics, please let me know by liking, commenting on, or sharing the Google+ and blog postings on
`https://www.google.com/+AartBik`
`http://aartbik.blogspot.com/`,
or simply drop me an email at `ajcbik@google.com`.

Source code of the examples and animated GIFs for some of the demos can be found by following the Micro-KIM link at `http://www.aartbik.com/`.