Tutorial 3

Using the KD30 Debugger

1 Introduction

Overview

The KD30 debugger is a powerful software tool that can greatly reduce the time it takes to develop complex programs on the M16C. Up until now, we have used KD30 as a means of running and testing our programs; but, as you will see, this merely scratches the surface of the debugger's capabilities.

KD30 has many features you can use to quickly locate and isolate faults in your programs. For example, you may:

- Set breakpoints in your code
- Step through code one instruction or statement at a time
- Examine the contents of memory and registers
- Evaluate C expressions / examine C variables at runtime
- View the disassembly of your C program

After completing this tutorial, you should be able to perform each of the tasks on the list above, and have an understanding of why and when you'd want to use each feature.

1 The Example Program – Sample3

The program "sample3" we use in this tutorial is very similar to "sample2" from the last tutorial. Once again we have a counter program, but the code has been enhanced

to cover the range 0-99 and display output on both LED1 and LED2.

The program uses two timers, TA0 and TA1, and two interrupt service routines, `TimerA0int()` and `TimerA1int()`.

- Timer A0 performs the same function as last time. TA0 is the clock, interrupting at a rate of 10Hz (100ms), responsible for incrementing the counter once per second.

- Timer A1 is responsible for multiplexing the display of LED1 and LED2; TA1 interrupts every millisecond, yielding a refresh-rate of 100Hz.

One limitation of the M16C development board is that the two 7-segment displays, LED1 and LED2, are both controlled via I/O port **p0**, meaning either LED1 or LED2 can be enabled, but not both at the same time. To work around this, we have introduced a new timer, TA1, to rapidly switch the output between LED1 and LED2, giving the illusion that both LEDs are simultaneously lit.

2 An Example Debugging Session

The first thing we want to do is initialize our debugging session.

- Start KD30 now. Do this via TM, or by launching KD30 from the Start menu.
- Choose File → Download → Load Module…
- Open the **sample3.x30** file from the resources\sample3 directory.
- Click the **Go** button to make sure the program was loaded and initialized properly. You should see the counter program running, displaying its output on both 7-segment displays.
- Hit **Stop** to stop the program running, then **Reset** to reinitialize KD30. We're ready to start now.

Note: If the debugger crashes or locks up at any point during the tutorial, use these steps to reinitialize KD30 and then continue with your debugging session.

2 The KD30 Interface

The first thing we're going to do is take a look at is the KD30 interface and how to use it to navigate around your source code.

The Program Window

The Program Window is the view you will make most use of while debugging programs. It allows you to view the source code and disassembly of your program, set and clear breakpoints, and navigate through your code.

```
Program Window [ncrt0.a30]
View | Source | Mix | Di...
Line  BRK PASS Source
00112           ;---------------------------------------------------------------------
00113                   .insf     start,S,0
00114                   .glb      start
00115                   .section          interrupt
00116           start:
00117           ;---------------------------------------------------------------------
00118           ; after reset,this program will start
00119           ;---------------------------------------------------------------------
00120  -                ldc       #istack_top,     isp        ;set istack pointer
00121  -                mov.b     #03h,0ah                    ;enable PRC1 and PRC0 bits
00122           ;       bset      1,0ah
00123  -                mov.b     #00h,04h                    ;set processor mode
00124           ;       bclr      1,0ah
00125  -                mov.b     #08h,06h                    ;set to no division mode on main cloc
00126  -                mov.b     #20h,07h
00127  -                mov.b     #00h,0ah                    ;disable PRC1 and PRC0 bits
00128
00129  -                ldc       #0080h, flg
00130  -                ldc       #stack_top,      sp         ;set stack pointer
00131  -                ldc       #data_SE_top,    sb         ;set sb register
00132  -                fclr      I
00133  -                ldintb    #VECTOR_ADR
00134  -                fset      I
00135
00136           ;=====================================================================
```

Figure - An example Program Window showing the M16C bootstrap code. The line highlighted in yellow indicates the current position of the Program Counter (PC) register.

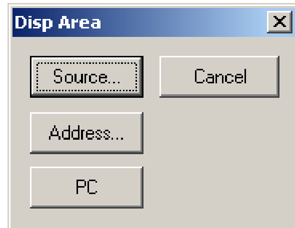
The line marked in yellow represents the current position in our program, i.e. the memory location pointed to by the Program Counter (PC) register. This is the point where execution will begin when you start your program running.
The three buttons at the top of the window, **Source**, **Mix**, and **Disassembly** allow you to switch between source-code view (useful when debugging C programs), a mix of source-code and disassembly (for examining what your C code compiles to), and straight disassembly (for the hardcore ASM purist…).

At this point in time, we are located at the beginning of the M16C bootstrap code; the procedure for initializing the microprocessor and special areas of memory such as the stack and heap, and transferring control to our main() function. The bootstrap code is written in assembly language, so the **Source** and **Mix** views offer little advantage over the **Disassembly** view for this particular section of code. If you're interested, the bootstrap code is contained in the file **ncrt0.a30**.

Now to find our **main()** function.

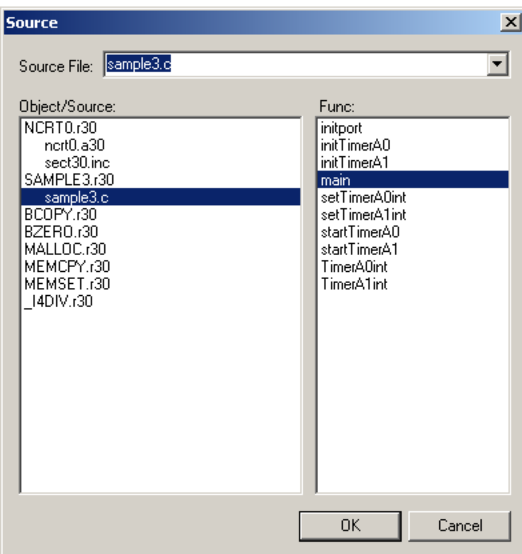
Navigating your Source Code

The **View** button allows us to navigate around our program.



Click **View** now to open the Display Area dialog. **Source** allows you to navigate to a position in a source code file. **Address** allows you to navigate to a specific memory address. **PC** takes you to the current address pointed to by the Program Counter.

- Click **Source…**

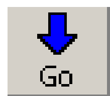


The source dialog allows you to browse your project by source file or object file; it displays a list of each function within the source file or object.

- Select **sample3.c** from the Source File drop-down menu to browse the function list for the sample3.c file.
- Select **main** from the list of functions in the right-hand pane and click OK.

3 Running your Program

Go



The **Go** button starts running your program; no surprises here. Go will execute your program until it reaches a breakpoint, or you manually stop the program using the **Stop** button.

Let's try this now:

- In the Program Window, click on the initport() function to move the cursor to the correct position.
- Click the **Break** button to set a breakpoint at this line. You will notice a 'B' appear in the BRK column to the left to indicate that a breakpoint has been set, and the line should be highlighted in red.
- Click **Go** to run the program.

The program will run for a little while, and then stop; highlighting the initport() line in yellow. The microprocessor has executed everything up until this point (the bootstrapping code we discussed earlier) and stopped executing before entering the initport() function. As you can see, the LEDs on the M16C development board are not lit, since we have suspended the program before writing any data to the I/O ports.

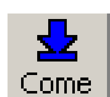
A word of caution on using breakpoints:

A breakpoint will persist until you remove it or KD30 is closed. If you're not careful, this can lead to the debugger behaving unpredictably.

If you modify, recompile and download a new version of your program to the development board, be sure to remove any breakpoints you have set beforehand. Your breakpoints are still assigned to the physical memory addresses of symbols from the *old* version of your program. Those symbols may no longer exist in the new version of your program, and, even if they do still exist, there is no guarantee they will reside at the same physical memory address as before, since the compiler is free to assign your code to another block of memory during the linking phase.

To quickly remove your breakpoints, use the **S/W Breakpoint** button. This will allow you to manage all your breakpoints at the click of a button, and remove them without having to search through the code.

21. Come



The **Come** button executes your program from the current position of the program counter (PC) to the statement selected by the cursor in the Program Window. The program will continue running until a breakpoint is hit or the statement selected by the cursor is reached.

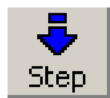
Using **Come** is a convenient way to execute your program up until a certain point in a file without having to manually set a breakpoint.

- Place the cursor on the setTimerA0int() function three lines down.
- Click **Come**.

Execution will stop at the setTimerA0int() function. At this point, the I/O ports have been initialized (though don't expect them to light up yet; no data has been written!), as have both TA0 and TA1.

22. Step Into / Step Over

The **Step** and **Over** buttons are useful for moving through a program one line, or statement, at a time.



The **Step** button executes the current statement of your program, automatically breaking before the next statement. Using **Step** on a function call or loop conditional statement will step *into* the function or loop.

The **Over** button performs a similar operation to the **Step** button, but treats C statements such as function calls and loops as atomic blocks. Using **Over** on a function call will execute the entire function and break on the next line of your C program. Using **Over** on a loop conditional statement will execute a single iteration of the loop, or break on the C statement following the loop should the loop condition evaluate to false.

Let's try the step feature now:

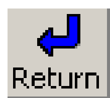
- Click **Step** to step into the setTimerA0int() function. You'll see that we've now moved into the setTimerA0int() function.
- Click **Step** to enter the function, again to execute the program statement, and

once more to leave the function.

The yellow line should now be highlighting the setTimerA1int() function. Now try step over:

- Click **Over** to execute the setTimerA1int() function. This will execute the entire function as if it were a single line of code.

23. Return



The **Return** button executes the program from the current position of the program counter (PC) until a return instruction is called and execution is returned to the current function's caller. **Return** tends to be most useful when you've accidentally stepped *into* a function you intended to step *over*.

4 Examining the Contents of Registers

The Register Window, shown below, allows you to view and modify the contents of the M16C registers and microprocessor flags.

- Select Basic Windows → Register Window from the KD30 menu to open the Register Window.

0 Bank - Register Window

Name	Value	Radix
PC	0F0021	Hex
R0	0000	Hex
R1	0F00	Hex
R2	0000	Hex
R3	0000	Hex
A0	0000000000011101	Bin
A1	0000010000101001	Bin
FB	0724	Hex
USP	0724	Hex
ISP	0A29	Hex
SB	0400	Hex
INTB	0FA000	Hex

IPL	U	I	O	B	S	Z	D	C
0	1	1	0	0	0	1	0	1

You can choose the radix for each individual register by right-clicking on the register name and selecting Hex, Dec or Bin. The screenshot to the left shows the address registers, A0 and A1, displayed in binary and the remaining registers displayed in hexadecimal. Double-clicking a register allows you to change the display radix as well as setting the contents of the register.

A register that has changed value since the last breakpoint is highlighted in red. Try stepping through a few lines of code and take note of changing register values.

Examining the contents of registers is most frequently used when you need to debug assembly language code.

5 Examining the Contents of Memory

The Memory Window

The Memory Window shows the contents of memory linearly (like one very large array), in blocks of 1, 2, or 4 bytes.

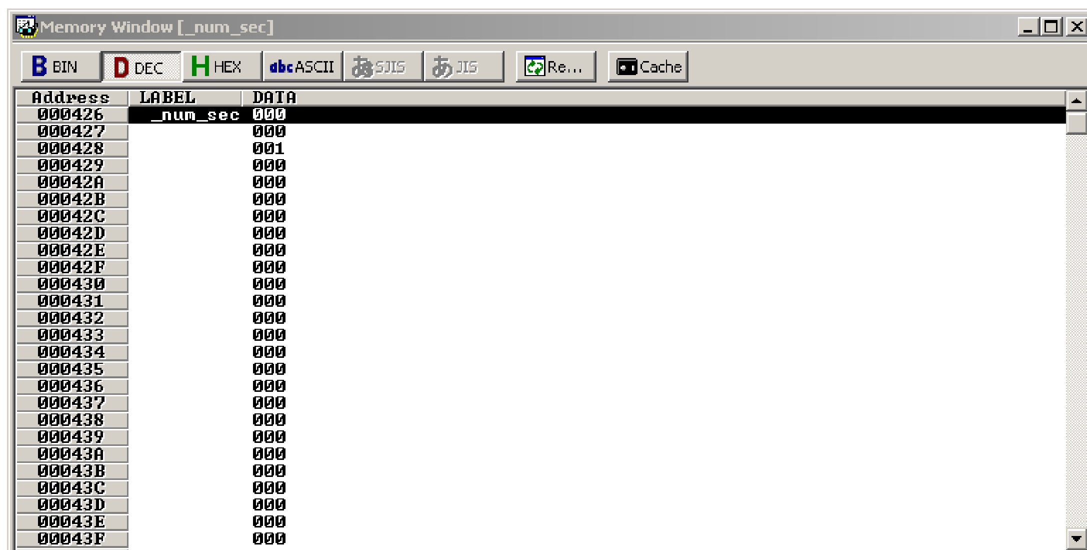


Figure - An example Memory Window displaying the range 0x000426 to 0x00043F in memory. The *num_sec* variable from our example program is located at address 0x000426.

The Address column on the far left shows the range of memory addresses currently being viewed. The Label column displays the name of any C symbol that resides within the address range; the **num_sec** variable from our example program is located at memory address 0x004026 in , for example.

Data can be displayed in bytes, words or lwords; and can be formatted as binary, decimal, hex, or as ASCII characters. The list of display options is available by right-clicking in the Memory Window.

The Memory Window is most useful for locating and displaying the contents of symbols, i.e. variables and functions, from your C programs. To give you an example of what can be done using the Memory Window, we will use it to snoop on the

num_sec variable from our example program.

Manually Watching a C Variable

- Hit the **Reset** button in KD30 to reinitialize the debugger.

The first thing to do is to set our breakpoint. We want to choose a position in the program that allows us to watch the **num_sec** variable change as the program is running. This change occurs in the Timer A0 interrupt service routine.

- Use the **View** button in the Program Window to navigate to the TimerA0int() function in the file sample3.c
- Position the cursor on the line that reads **num_sec++**
- Hit **Break** to set a breakpoint at this line.

Setting a breakpoint here will allow us to examine the value of **num_sec** before, and after, the variable is incremented.

Now, set up the Memory Window to watch the **num_sec** variable:

- Open the Memory Window if you haven't already. Select BasicWindows → Memory Window
- Now, double-click on any address in the Address column to the left. This opens the Display Address dialog.
- Enter **_num_sec** into the Address field (don't forget the leading underscore!). Hit OK. You should see the label **_num_sec** displayed in the Label column on the left-hand side.
- Change the display to **Byte**; decimal radix. Your Memory Window should look more or less like the screenshot at the beginning of this section.

We're ready to start debugging.

- Click **Go** to start the program running.

The program will break in the TimerA0int function. Take a look at the value of the **num_sec** variable in the Memory Window. Its value should be zero.

- Click **Step**. We have now incremented the value of **num_sec**.
- Click **Go** to run the program again. Take a look at the LEDs on the development board while the program is running (be quick; you only have 1 second to see the display before the program breaks again).

Repeat this process a couple of times and watch the **num_sec** variable and LED display.

If you look closely, you'll notice the display doesn't look quite as 'solid' while the program is running. Why do you think this is?

The Dump Window

The Dump Window shows a hexadecimal dump of the contents of memory, familiar to anyone that has previously used a hex-editor.

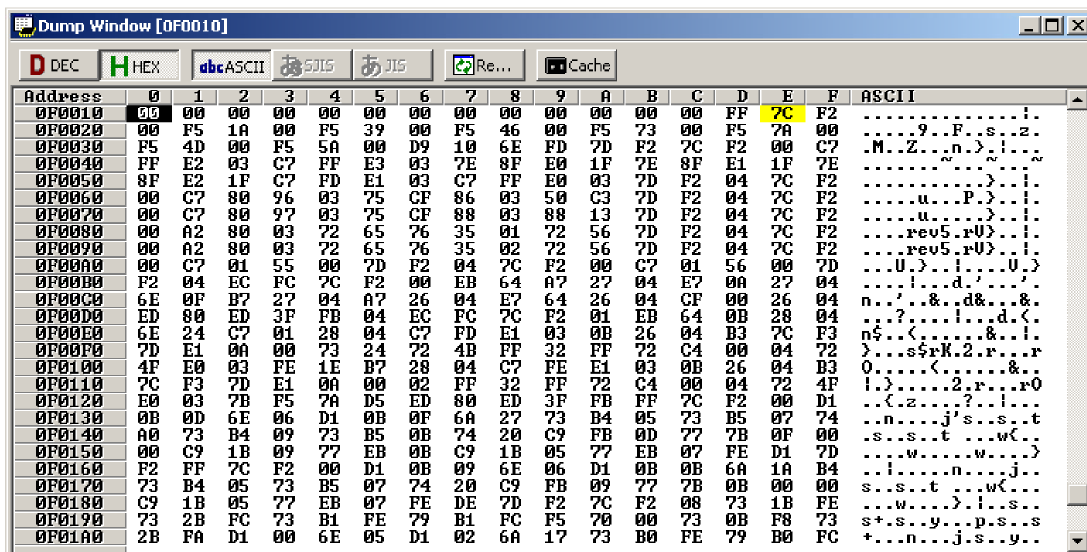


Figure - A hex-dump of the main() function from our example program. The main() function begins at address 0x0F001E, highlighted in yellow.

The Dump Window is most useful for editing the contents of memory. In particular, you can use the Dump Window to manually set the contents of I/O ports and control registers while your program is running. To illustrate, we will use the Dump Window to toggle LED1 and LED2 on the development board.

- Hit **Reset** to reinitialize KD30
- Remove breakpoints from the last run using the **S/W Breakpoint** tool.
- Navigate to the TimerA1int() function. A code snippet is shown below.

```
void TimerA1int(void)            // interrupt function
{
       static unsigned char led_select = 0;
       unsigned char offset;

       _asm("fset I");            // enable interrupts
       if (led_select == 0)
              {
              led_select = 1;
              p1 = 0xFD;          // enable LED2 and disable LED1
              offset = num_sec % 10;
              p0 = leddigit[offset];     <---- INSERT BREAKPOINT HERE
              }
       else
              {
              led_select = 0;
              p1 = 0xFE;          // disable LED2 and enable LED1
              offset = num_sec / 10;
              p0 = leddigit[offset];     // display quotient
              }
}
```

- Insert a breakpoint on the line indicated in the code snippet.
- Click **Go** to run the program up to our breakpoint.
- Click **Step** to execute the port write statement. A zero should now be displayed on LED2.

There are two things to take note of here: firstly, I/O port **p1** is responsible for selecting LED1 or LED2, using values 0xFE or 0xFD; and secondly, I/O port **p0** receives the data byte to be written to the selected LED. If we can find the locations of **p0** and **p1** in memory, then we can manipulate the 7-segment display by hand.

- Open the Dump Window if it is not already open. Select BasicWindows → Dump Window.
- Double-click on a memory address in the Address column. Enter the value **3E0** in the Address field. This is the memory address of port **p0**.
- Double-click the value at address 0x0003E0 (it should be C0). The Set dialog should appear
- Enter the value **89** into the Value field and click OK. The display on LED2 should have changed to an ‘H’.
- Now enter the following digits in sequence, one digit at a time, and observe the display: 86, C7, C3, and C0.

Now to try swapping the LEDs. This time, we have to write data to port **p1** instead, since it is port **p1** that determines which of the two LEDs is currently displayed.

- Use the Set dialog to cycle the value of memory location 0x0003E1 between 0xFD and 0xFE a couple of times.

LED goes left… LED goes right… LED goes left… LED goes right…

Ok. You get the idea.

While this might seem to be a trivial demonstration, the ability to set ports and registers at runtime can save you a *lot* of debugging time when you begin working with more complex programs; particularly when tweaking hardware to find optimal settings. For example, if you're working with a timer, you can experiment with various time periods from within the debugger, rather than having to recompile and download a modified version of your program every time you require a change.

A complete list of all the M16C special registers and ports, and their addresses, can be found in the **sfr62.h** header file.

6 The C Watch Window

In section we used the Memory Window to watch the value of a C variable while the program was executing. In this section, we will introduce an alternative method you can use to watch C variables: The C Watch Window and its derivatives.

The **Local Window** displays the contents of variables that are local to the function you are currently executing. The Local Window is particularly useful for viewing the contents of variables that have automatic scope, and in fact, without a feature like the Local Window, it would be very difficult to monitor the contents of automatic variables.

The **Global Window** displays the contents of global variables; that is, variables in your program that lie outside the scope of any function.

The **File Local Window** displays the contents of variables local to the current C source file.

The **C Watch Window** allows you to enter custom expressions, in C, that will be evaluated and displayed at runtime.

Using Watch Windows

Let's take a look at how to use the Local and Global Windows now:

- Hit **Reset** to reinitialize KD30
- Remove any existing from previous runs. We don't require any breakpoints for this example.
- Open the **Local Window** and the **Global Window** from the BasicWindows menu.

Take a look in your Local and Global windows now:

The Local Window should contain no variables. This is exactly what we expect; the program counter is positioned at the very beginning of the M16C bootstrap code, and we are not 'inside' any C function at this stage of the program.

The Global Window should contain two variables: the **leddigit** array, containing data for displaying digits on the 7-segment display; and **num_sec** that stores the number of seconds elapsed. Don't worry too much if the value of **num_sec** is not zero as you might have expected. If this is the case, the value displayed here is the value **num_sec** reached on the last run of the program. The bootstrap code, that we haven't got around to executing yet, will initialize the contents of RAM before our main() method is called.

- Navigate to the beginning of the main() function.
- Position the cursor on the initport() function call and click the **Come** button.

The bootstrap code has now run. If you check the value of **num_sec** in the Global Window, it should now be zero. Since our main() function has no local variables, the Local Window is still empty.

- Click **Go**. Let the program run until the LED counter has reached 5 or so, then stop the program using the **Stop** button.

Check the Global Window; the value of **num_sec** should now have changed.

Now let's take a look at some local variables:

- Locate the TimerA0int() function.
- Position the cursor somewhere within the function. The line that increments `one_hundred_msec_count` is a good choice.
- Click **Come** to run the program to this position.

The Local Window should be displaying the value of the `one_hundred_msec_count` variable.

- Locate the TimerA1int() function.

- Position the cursor somewhere within the function.
- Click **Come** to run the program.

The Local Window now displays the contents of the `led_select` and `offset` variables.

Thanks to Jacob Hart for detailed and extensive help in the preparation of this tutorial