

## Lab 2 Software Debugging

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

- Goals**
- to develop software debugging techniques
    - Functional debugging (static)
    - Performance debugging (dynamic or real time)
    - Profiling (detection and visualization of program activity)
  - to manipulate bit streams with Huffman encoding and decoding.
- Review**
- Valvano Section 2.11 on debugging,
  - Ports J, K, and T of the MC68HC812 Technical Data Manual,
- Starter files**
- SIMPLE.C, BITFIFO.C, BITFIFO.H, HUFF.C, HUFF.UC, HUFF.IO, HUFF1.C

### Background

When information is stored or transmitted there is a fixed cost for each bit. Data compression and decompression provide a means to reduce this cost without loss of information. If the sending computer compresses a message before transmission and the receiving computer decompresses it at the destination, the effective bandwidth is increased. In particular, we will study ways to process bit streams using Huffman encoding and decoding. A typical application is illustrated by the following flow diagram.

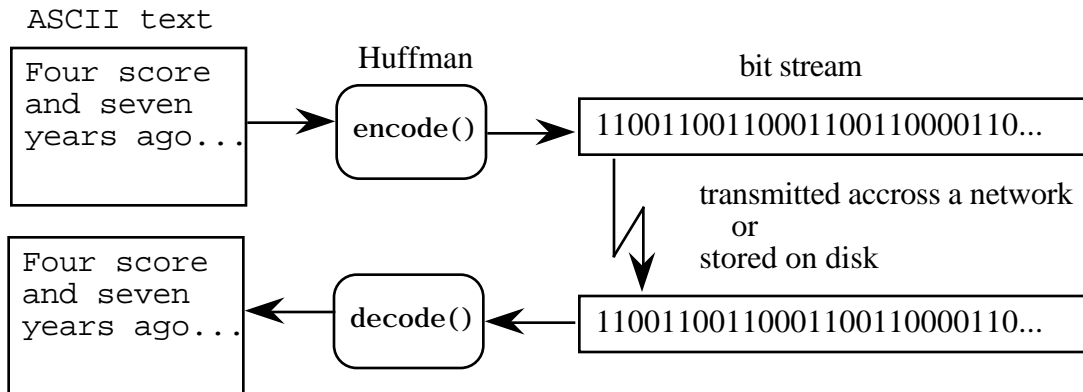


Figure 2.1. Data flow diagram showing a typical application of Huffman encoding and decoding.

The Huffman code is similar to the Morse code in that they both use short patterns for letters that occur more frequently. In regular ASCII, all characters are encoded with the same number of bits (8). Conversely, with the Huffman code, we assign codes where the number of bits to encode each letter varies. In this way, we can use short codes for letter like "e s i a t o n" (that have a higher probability of occurrence) and long codes for seldom used consonants like "j q w z" (that have a lower probability of occurrence). To illustrate the encode-decode operations, consider the following Huffman code for the letters M, I, P, S. S is encoded as "0", I as "10", P as "110" and M as "111". We can store a Huffman code as a binary tree.

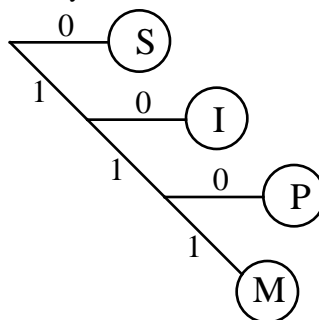


Figure 2.2. Huffman code for the letters S I P M.

If "MISSISSIPPI" were to be stored in ASCII, it would require 10 bytes or 80 bits. With this simple Huffman code, the same string can be stored in 21 bits.



Figure 2.3. Huffman encoding for MISSISSIPPI.

Of course, this Huffman code can only handle 4 letters, while the ASCII code has 128 possibilities, so it is not fair to claim we have an 80 to 21 bit savings. Nevertheless, for information that has a wide range of individual probabilities of occurrence, a Huffman code will be efficient.

### Preparation (do this before lab starts)

1. Download the `SIMPLE.C` file from the class web site. Compile this program and print out the entire listing file. Circle on the listing the assembly code that implements the `d=a+b+c;` line. Also print out the associated map file. Hand execute this program up to and including the `d=a+b+c;` line within the function (i.e., show the values in Registers D, X, Y, and SP after each instruction). Also, draw a stack picture at this point. Include the 16-bit values on the stack as well as the values in Registers D, X, Y, and SP.
2. Download the Huffman files `HUFF.C`, `BITFIFO.H`, `BITFIFO.C` files from the class web site. Make a printout of these files. Compile, download and run them. There is a bug in the function `PutBit()`. It works for a while, then crashes when the pointers wrap around. For now, use the programs as they are, and as part of the lab you will be evaluating various methods to visualize the bug. Edit, compile, assemble, download, and run this system with the bug observing its behavior.
3. In this part, we will study the execution speed of the routine `PutBit()` without the bug. Make a copy of the `BITFIFO.C` file, and edit the `PutBit()` function removing the bug. Compile, download and run the corrected version. Edit the assembly listing (e.g., `HUFF.LST`) file leaving just the assembly code that implements `PutBit()`. Print out this assembly listing of the `PutBit()` function. Please don't print the entire listing file, just the part that implements `PutBit()`. Look up the cycle count for each instruction in the `PutBit()` function, and record it on the printout. Estimate the total execution time in  $\mu$ s for the `PutBit()` function. The 8 MHz E clock means there are 125 ns/cycle.

### Procedure (do this during lab)

#### 1. Functional Debugging: Verify input/output parameters

##### 1.A. Single Stepping or Trace

Using the `SIMPLE.LST` assembly listing file as a reference, single step the `SIMPLE.C` program. It may also help to have a printout of the corresponding map file. To single step this program with the Kevin Ross BDM, use the following commands

```
reset
load simple.s19
er pc f000
t
t
```

Pause at after the `d=a+b+c;` line within the function and verify your hand calculations you performed for the preparation. Continue executing the `t` command until the Adapt812 red LED turns on (about 53 times in all).

##### 1.B. Breakpoints without filtering

The first step of debugging is to *stabilize* the system with the bug. In the debugging context, we stabilize the problem by creating a test routine that fixes (or stabilizes) all the inputs. In this way, we can reproduce the exact outputs over and over again. Once stabilized, if we modify the program, we are sure that the change in our outputs is a function of the modification we made in our software and not due to a change in the input parameters. Modify the `main()` program to stabilize the bug in `PutBit()`. One way to stabilize the bug is to write a main program that calls `InitBit()` followed by multiple calls to `PutBit()`. An example of this can be found in the `Huff1.C` program from the web. In this section, we won't correct the error, but rather we will investigate various methods to illustrate it. A crude way to set breakpoints is to define

```
#define bkpt asm(" bgnd");
```

and insert `bkpt` at the places you wish to break. The software must be recompiled and reloaded after every breakpoint is added or removed. In addition, the software must be started from the BDM debugger using the commands

```
reset
```

```
load huff1.s19
g f000
```

After a breakpoint, registers and memory can be viewed or changed. To single step you can execute `t`. To restart execution after the breakpoint, type `g`. To remove a breakpoint, you have to edit, compile, and load.

Set a breakpoint right before the line containing `TempPutPt.WPt++` within the `PutBit()` function. Run the stabilized system, and at each breakpoint record the value of `TempPutPt.WPt` (it can be found on the stack.)

### 1.C. Conditional breakpoints

One of the problems with breakpoints is that sometimes we have to observe many breakpoints before the error occurs. One way to deal with this problem is the conditional breakpoint. Add a global variable called `count` and initialize it to zero in the ritual. Add the following conditional breakpoint to the beginning of `PutBit()`. And run the system again (you can change the 32 to match the situation that causes the error.)

```
if(++count==32)
    bkpt
```

Notice that the breakpoint occurs only on the 32<sup>nd</sup> call to `PutBit()`.

### 1.D. Instrumentation: print statements

Using the editor, add print statements (use should use the routines from `SCI12.H` and `SCI12.C`), that we will call debugging instruments, to `PutBit()` that illustrate the programming error when `main()` is executed. A key to writing good debugging instruments is to provide for a mechanism to reliably and efficiently remove them all when the debugging is done. Consider the following mechanisms as you develop your own unique debugging style:

- Place all instruments in a unique column (e.g., first column.), so that the only codes that begin in this column are debugging instruments.
- Define all debugging instruments as functions that all have a specific pattern in their names. For example, make all debugging functions begin with “debug”. In this way, the find/replace mechanism of the editor can be used to find all the calls to the instruments.
- Define the instruments so that they test a run time global flag. When this flag is turned off, the instruments perform no function. Notice that this method leaves a permanent copy of the debugging code in the final system, causing it to suffer a runtime overhead, but the debugging code can be activated dynamically without recompiling. Many commercial software applications utilize this method because it simplifies “on-site” customer support.
- Use conditional compilation (or conditional assembly) to turn on and off the instruments when the software is compiled. When the compiler supports this feature, it can provide both performance and effectiveness.

### 1.E. Instrumentation: dump into array without filtering

One of the difficulties with print statements is that they can significantly slow down the execution speed in real time systems. Many times the bandwidth of the print functions can not keep pace with the existing system. For example, our system may wish to call `PutBit()` 1000 times a second (or every 1 ms). If we add print statements to `PutBit()` that require 50 ms to perform, the presence of the print statements will significantly affect the system operation. In this situation, the print statements would be considered extremely intrusive. Another problem with print statements occurs when the system is using the same output hardware for its normal operation, as is required to perform the print function. In this situation, debugger output and normal system output are intertwined.

To solve both these situations, we can add a debugger instrument that dumps strategic information into an array at run time. Write a debugger instrument that saves the current values of the strategic parameters at the end of each call to `PutBit()`. Write another routine that dumps the contents of the array at the end of `main()`. Make a print out of the run, again circling the situation where the first error occurred. One of the advantages of dumping is that the 6812 BDM module allows you to visualize global memory even when the program is running. So this technique will be quite useful in systems with a background debug module (the 68332 has one too.)

### 1.F. Instrumentation: dump into array with filtering.

Consider what would have happened in the previous section if the bit string were 10000 bits! One problem with dumps is that they can generate a tremendous amount of information. If you suspect a certain situation is causing the error, you can add a filter to the instrument. A filter is a software/hardware condition that must be true in order to place data into the array. In this situation, if we suspect the error occurs when the pointer nears the end of the buffer, we could add a filter that saves in the array only when

```
pt.WPt points to Fifo[maxsize-2]
```

Modify your instrument to add an appropriate filter. Again make a print out of the run, circling the situation where the first error occurred.

**1.G. Monitor using the LED display** *No action is required on your part for this section.*

Another effective tool for real time applications is the monitor. A monitor is an independent output process, somewhat similar to the print statement, but one that executes much faster, and thus is much less intrusive. The LCD display you will interface in a subsequent lab can be an effective monitor for small amounts of information. The Adapt812 has an LED on Port T bit 6. Your software could toggle this LED to let you know the program is running. This is an example of a BOOLEAN monitor.

One could classify the combination of the dump and the real time visualization of the dump contents with the BDM as a monitor.

**2. Performance Debugging: Execution Speed****2.A. Instrumentation measuring with an independent counter, TCNT**

In the preparation, you estimated the execution speed of the "corrected" `PutBit()` routine by counting instruction cycles. This is a tedious, but accurate technique on a computer like the 6812 (when running in single chip mode). It is accurate because each instruction (e.g., `ldd 4,x`) always executes in exactly the same amount of time. Cycle counting can not be used in situations where the execution speed depends on external device timing (e.g., consider how long it takes to execute `InChar`.) If the 6812 were to be running in expanded mode, the time for each instruction would depend also on whether it is accessing internal or external memory. On more complex computers, there are many unpredictable factors that can affect the time it takes to execute single instructions, many of which can not be predicted *a priori*. Some of these factors include an instruction cache, out of order instruction execution, branch prediction, data cache, virtual memory, dynamic RAM refresh, DMA accesses, and coprocessor operation. For systems with these types of activities, it is not possible to predict execution speed simply by counting cycles using the processor data sheet. Luckily, most computers have a timer that operates independently from these activities. In the 6812, there is a 16 bit counter, called TCNT, which is incremented every E clock. The MC68HC812 has a prescaler that can be placed between the E clock (8 MHz) and the TCNT counter. It automatically rolls over when it gets to \$FFFF. If we are sure the execution speed of our function is less than (65535 counts), we can use this timer to directly measure execution speed with only a modest amount of intrusiveness. Let `First` and `Delay` be unsigned 16 bit global integers. The following code will set the variable `Delay` to the execution speed of `PutBit()`.

```
First=TCNT;
PutBit(0);
Delay=TCNT-First-12;
```

The constant "12" is selected to account for the time overhead in the measurement itself. In particular, run the following,

```
First=TCNT;
Delay=TCNT-First-12;
```

and adjust the "12" so that the result calculated in `Delay` is zero. Use this method to verify the general expression you developed as part of the preparation.

Collect execution times for the function 1) as is, 2) with debugging print statements, and 3) with debugging dump statements. For the dump case, you are measuring the time to store into the array and not the time to print the array on the screen. The slow-down introduced by the debugging procedures defines its level of intrusiveness.

**2.B. Instrumentation Output Port.**

Another method to measure real time execution involves an output port and an oscilloscope. Connect a 6812 output bit to your scope. Add a debugging instrument that sets the output bit to one just before calling `PutBit(0)`; . Then, set the output back to zero right after. Remember to set the port's direction register to 1, e.g., `DDRT=0xFF`; If we put the instruments inside `PutBit`, we will measure the speed of the calculations and neglect the time it takes to pass parameters and perform the subroutine call. On the other hand, in a complex system this method allows you to visualize each call to `PutBit`, regardless from where it was called. For this lab, stabilize the input, and repeat the operation in a loop, so that the scope can be triggered. Compare the results of this measurement to the TCNT. Discuss the advantages and disadvantages between the TCNT and scope techniques. E.g.,

```
while(1){
    PORTT |= 0x01;    /* Port T bit 0 goes high */
    PutBit(0);
    PORTT &= ~0x01;   /* Port T bit 0 goes low */
    GetBit(&pt); }    /* remove so Fifo doesn't get full */
```

### 3. Profiling: Real time execution sequence

The objective of this part is to develop software and hardware techniques to visualize program execution in real time. In a subsequent lab, you profile the execution of a multiple thread software system to detect reentrant activity. But in this lab, we will study the execution pattern of the `decode()` function.


#### 3.A. Profiling using a software dump to study execution pattern

In this section, you will develop software instruments that study the execution pattern of the function `decode()`. Add 4 or 5 instruments to the function `decode()` that record:

location within `decode()` that the instrument was placed (e.g., PC value);

current value of a strategic program parameters, like `hpt` and `(*spt)`.

The debugging instrument should save in an array (like a dump). If the debugging instrument is implemented as a function you can read the return address off the stack (the place from that it was called). Write as much of the function as you can, compile it, and look at where on the stack the PC is located. Be careful not to alter the program function. Except for a modest increase in the execution time, your instrument should not modify the operation of `decode()`.

Run `decode()` with an example of 3 or 4 characters or 7-12 bits. Printout the C source code of the function `decode()`. After `decode()` has been executed, printout the results of the debugging instruments. On the source code listings draw "tail to head" arrows (e.g., ) illustrating the execution pattern. There should be at least 15 arrows. If the data you collect is confusing, repeat the experiment with more (or less) instruments.

#### 3.B. Profiling using an Output Port

In this section, you will construct a hardware/software combination to visualize program activity. Add four software instruments that set two output port bits to 00 01 10 11 respectively. Again place these instruments at strategic places in the function `decode()`. Stabilize the system so that `decode()` is called over and over. Connect 6812 output pins to a dual channel scope and observe the program activity. If the signals you are confused, move the instrument to a more strategic location.

#### 3.C. Thread Profile *No action is required on your part for this section.*

When more than one program (multiple threads) is running, you could use this technique to visualize the thread that is currently active (the one running). For each thread, we assign an output pin. The debugging instrument would set the corresponding bit high when the thread starts and clear the bit when the thread stops. We would then connect the output pins to a multiple channel scope to visualize in real time the thread that is currently running.

### Checkout

You should be able to demonstrate:

1. Part 1.F. Instrumentation dump into array with filtering.
3. Part 2.B. Instrumentation Output Port.
4. Part 3.A. Profiling using a software dump.
5. Part 3.B. Profiling using an output port.

### Hints

- 1) Do not type these software examples, download the latest versions from the web (ask your TA how).
- 2) If an assignment is unclear, ask your TA for a clarification.

```
/* *****simple.C***** */
#include "HC12.H"
int function(int a, int b, int c){ int d;
    d=a+b+c;
    return d;}
int e;
void main(void){ int i;
    COPCTL=0x00;    // disable COP
    DDRT=0xFF;     // Port T is output
    e=function(1, 2, 3);
    while(1){ i++;
        if(i>0)
            PORTT|=0x40;    // LED is on
        else
            PORTT&=~0x40;   // LED is off
    }}
}
```