Handout for EC331 Lab #3:
-- ASCII-Encoded Keyswitch Entry Terminal

Jianjian Song and Keith Hoover
Department of Electrical and Computer Engineering
Rose-Hulman Institute of Technology
Revised September 20, 2009

1 Objectives

(1) To learn modular design and development of software.

(2) To use 9S12C128 hardware timer to generate variable time delays.

(3) To use hardware-timed delay for debouncing an SPST switch.

(4) To write a routine for 9S12C128 to scan an X-Y matrix keypad.

(5) To learn how to use a lookup table for data conversion.

(6) To program the 9S12C128’s serial communications interface module (SCI) for asynchronous serial data communication.

2 Deliverable

Each lab group must submit a written memo-style lab report which refers to six different attachments, where each attachment corresponds to one of the five “Lab Report” sections labeled “Lab Report Part 1” through “Lab Report Part 5”, which are scattered throughout this lab handout. (See the table of contents.) The central theme of this lab project is to show that each module was first written (as a subroutine) and separately tested and debugged (by calling it from a very simple test program) ***before*** the final step, in which all modules are integrated into the final project.

3 Reference Materials and Equipment

3.1 References

(1) TIM16B8 Timer Block User Guide (***S12TIM16B8CV1.pdf***)
(2) MC9S12C Family Device User Guide (***9S12C128DGV1.pdf***)
(3) HCS12 Microcontrollers Reference Manual (***S12CPUV2.pdf***)

3.2 Equipment

CSMB12C128 evaluation board and a 16-key 4 x 4 row-column matrix keypad (available from the instrument room.)

4 Introduction

This project is intended to illustrate how a relatively large project, in this case, the design of a software-debounced keypad entry system with asynchronous serial (RS232) output, can be approached in a top-down, modular, “divide-and-conquer” fashion.

Along the way, you will gain experience working with the 9S12C128’s hardware timer, software switch debouncing, key depression identification from a row-column keypad, the use of a lookup table, and the programming of the 9S12C128’s asynchronous serial communications interface (SCI) module.

5 Modular Programming

Before we get started with this project, let us discuss the concept of modular programming.

5.1 Software modules

A software module is very much like a hardware module or integrated circuit chip, in that it is an easily detachable and separately testable section of code that has just one entry point and one exit point.

In this project, each module will be written as a subroutine. A separate project will be created to test each subroutine using a "short and simple" test program that calls that subroutine under test and exercises it sufficiently to verify its correct operation. Finally, once all of the component subroutine modules have been individually validated, they will be integrated together via a main program that calls each subroutine in turn.

5.2 Top-down modular design

The steps in the top-down modular system design (this applies to *both* hardware and/or software design) are listed below:

(1) Precisely and fully specify the overall design. When a need arises for a new product, many "gaps" usually exist in the initially (somewhat vague) product description. Many secondary design decisions usually have to be made in order to "fill the gaps".

(2) Intuitively break the overall design into smaller subsystems (modules), which are easily designed, debugged, and modified separately from each other. Furthermore, each module should be able to interconnect with the other modules in a clean and straightforward way that requires the fewest possible number of "interconnections".

(3) Develop precise and detailed specifications for each module.

(4) Implement each module exactly as specified. (Perhaps several designers will implement several modules concurrently.) New designers sometimes have trouble following this step. Although they are certainly encouraged to be as clever and creative as possible as they *internally design* the modules, they *MUST NOT* creatively modify any of the external module interface specifications in any way. After all, their modules are going to have to integrate with other modules that have been created by other designers.

(5) The source (assembly or C) code for each module (usually a subroutine) must be fully described, along with revision number and author, in a "***header comment block***". Many companies have a standard header block format that must be followed by any programmers who work for the company. This guarantees that all modules are fully documented right in their source code, permitting these modules to be used easily in future applications as well as in the present one.

(6) Fully debug and then test each module SEPARATELY until each module fully complies with the specifications that were assigned to it.

(7) Integrate all modules into a complete system. Then test to see that the original overall system specifications have been fully met. If not, "loop back" to Step 4.

5.3 Documenting a software module: the header block

Each software module that you produce should be accompanied by a software interface definition. This definition typically appears as a header block at the beginning of the module's source code. The header block consists of a series of comment lines (perhaps set off using rows of asterisks). In general, a well-documented module should contain a header block that includes the following information:

(1) The program (or subroutine) name.

(2) Description of its intended purpose and possible application.

(3) Author's name, phone number, and company location and job site.

(4) Date of initial writing.

(5) For each revision made: revision number, descriptions of revision made and date of revision.

(6) Text or journal reference(s) to algorithm(s) used in the module.

(7) Description of all input data needed, and how this input data is to be sent to the module. (There are at least three possibilities: (a) in specified machine registers, (b) in specified RAM locations, and (c) on the stack.)

(8) Description of output data produced by the module; how each resulting output datum relates to the input data, and where each of the output data can be found after the module is executed.

(9) Statement of potential problems or side effects of using this software module. Indicate which, if any, internal machine registers are modified (clobbered) by the module.

6 Step 1: Write a variable-time-delay subroutine "wait_y_4us"

In this section you will use one of the 9S12C128's "Output Compare" registers to write a variable time delay subroutine that meets the software interface specification that appears in the next section.

6.1 Software interface specification for subroutine "wait_y_4us"

Write a 9S12C128 assembly-language subroutine named "**wait_y_4us**" which, when called, will make use of the 16-bit hardware timer (binary up counter) register (TCNT) in conjunction with one of the "Output Compare***" timer channel*** registers (say TC2) to "waste time", or delay, for N_y*2 microseconds before returning control back to the calling program. No registers may be altered by this subroutine, including the Y register.

The symbol N_y represents an unsigned 16-bit integer (input argument) that is placed in the Y register before the subroutine is called. This delay routine should use the "software polling" mode of operation; it should NOT be made interrupt-driven. (We will work with interrupt-driven output compare applications in Lab Project 4.)

The next section outlines how to use the output compare feature of the 9S12C128 to obtain reasonably accurate delay times in your delay subroutine.

6.2 Using the output compare feature of the 9S12C128's hardware timing module

Read about the function and programming of Output Compare registers in Section 4.4 of the ***TIM16B8 Block User Guide*** (***S12TIM16B8CV1.pdf***). For the sake of brevity, I will refer to this document as the "TIM User Guide" from now on.

Referring to Fig. 4-1 of the TIM User Guide, it should be evident that, in order to use the 16-bit Output Compare register TC2, which is accessible at two adjacent 8-bit addresses, TC2H:TC2L, you must first turn on the timing system by setting the timer enable (TEN) bit (Bit 7) of the TSCR1 (Timer System Control Register 1) to 1. Setting TEN to one will enable the free-running 16-bit timer (binary up-counter) TCNT, allowing it to begin counting (See Section 3.3.6 of the TIM User Guide).

Since TCNT is 16 bits long, and the 9S12C128 is an "eight bit" microcontroller, meaning that there can only be 8 bits at each memory location, TCNT must be broken up into two adjacent (8-bit) registers (just as with any 16-bit register in the 9S12C128), with the high (most significant) 8 bits of TCNT in register "TCNTHi", and the lower 8 bits of TCNT in register "TCNTLo". We will allow TCNT to "free run", continuously counting from 0x0000 up to 0xFFFF, and then wrapping back to 0x0000 and repeating the cycle endlessly (until the microcontroller is reset).

The rate at which TCNT is incremented (clocked) is selected by the prescaler bits, PR2:PR0. These prescale bits may be found in the bottom 3 bits of the Timer System Control Register 2 (TSCR2), as explained in Section 3.3.11 of the TIM User Guide. Your subroutine should set these prescale bits to ***%011*** (see Table 3-4 in the TIM16B8 Block User Guide) so that the 2 MHz internal bus clock frequency will be "prescaled" (divided) by 8, and then used to clock the 16-bit up counter, TCNT = TCNTHi:TCNTLo. Therefore, we may assume that TCNT increments once every 8 / 2 MHz = 4 µs. Note that the bus clock frequency is ½ the 4 MHz external crystal resonator frequency.

You must also set Bit 2 (TIOS2) of the TIOS (Timer Input Capture/Output Compare Select) register to configure the TC2 (16-bit Timer Channel 2) register (Section 3.3.1 of the TIM User Guide), to function as either an "***Output Compare***" register as opposed an "***Input Capture***" register, as explained in Section 4.4 of the TIM User Guide. .

Since TC2 is configured as an "***Output Compare***" register, the contents of the TC2 register are continuously compared by hardware with the contents of the free-running TCNT register, and a flag (bit 2 of TFLG1) is set to one when TCNT = TC2, and it stays set until the program writes a "1" to the flag (bit 2 of TFLG1) to reset it to zero. This flag bit can either be read in a loop, waiting for the flag to be set to one, or it can be configured to interrupt the processor when it becomes set. Thus the "Output Compare" mechanism permits us to generate precisely timed delays in a program. We will use this mechanism in its non-interrupt driven mode in this lab project.

If TC2 had instead been configured as an "***Input Capture***" register (by clearing Bit #2 of the TIOS register), the TC2 register is loaded with the value of TCNT at the precise time that a specified transition occurs (either rising edge, falling edge, or either edge) on the associated Port T input pin, PT2; and at this same instant of time, the associated flag bit (bit 2 of TFLG1) is set. Thus the "Input Capture" mechanism allows us to determine the precise times between successive edges (transitions) in a digital input signal. We will work with the Input Capture mechanism in Lab 4.

Next you may want to clear Bit 2 in the Timer Interrupt Enable (TIE) register to ensure that TC2-related interrupts are disabled (See Section 3.3.1 of the TIM User Guide). However, because the TIE register is zeroed out of reset, you should not strictly have to perform this step.)

Next you should read the present value of the 16-bit TCNT free running timer, which is accessible at two consecutive addresses, TCNTHi (high byte) and TCNTLo (low byte). (See Section 3.3.5 of the TIM User Guide.) Because we know that the TC2 flag in the TFLG1 register will be set as soon as the TCNT register counts up to a value that equals the value loaded in the TC2 register (Section 4.4 of the TIM User Guide), you simply need to add the value in the 16-bit Register Y (which contains Ny, the number of 2 µs time increments we desire to wait) to the current time value found in TCNTHi:TCNTLo. Then store the result in the TC2 Output

Compare Register at TC2Hi:TC2Lo. This act "*schedules*" an output compare event to occur at a time equal to Ny*2 µs from the present time, thereby achieving the desired delay.

Immediately after you schedule the new output compare event, you must take action to ***CLEAR*** the TC2 Output Compare flag bit by writing a "1" (***Not a "0"!***) to the position of that flag bit (Bit #2) in the Main Timer Interrupt Flag (TFLG1) register, since TC2 flag has probably already been set from a previous time that the TCNT register "counted past" whatever (garbage value) was previously held by the TC2 register before the desired output compare event was scheduled. Note from Section 3.3.12 in the TIM User Guide that that writing a zero to a bit position in TFLG1 will NOT change that flag. Thus the instruction ***MOVB #4, TFLG1*** will clear the TC2 flag, but it will NOT affect the state of any of the other timer interrupt flags ***(***See Section 3.3.12 of the TIM User Guide). This is important if other output compares are simultaneously being used in a program to perform other timing functions.

Finally, your program must enter a loop that waits until TCNT counts up to a value that equals the contents previously stored in the TC2 Output Compare register. This event causes the TC2 flag in the TFLG1 register to be set to 1 by the Output Compare hardware at the point in time when TCNTHi:TCNTLo has counted up to the value stored at TC2H:TC2L. This happens after Ny*2 µs have elapsed.

Note that the Timer Control Registers 1 and 2 (TCTL1 and TCTL2) allow output compare events to cause automatic changes on the corresponding PORT T I/O pin. (See Section 3.3.8 and Table 3.2 of the TIM User Guide). However, in this application we do not want to activate this feature, allowing the PORT T pins to be used for simple parallel I/O functions. Note that TCTL1 and TCTL2 registers are cleared out of reset, and according to Table 3-2 of the TIM User Guide, this will disable the output compare function on the PORT T pins, as we desire. Therefore your subroutine can either ignore TCTL1 and TCTL2, or you may want to set them to zero up near the beginning of the subroutine, just to be sure that an output compare event will *not* affect the PORT T pins.

Your subroutine ***MUST*** preserve the contents of the A, B, X, Y microcontroller registers if they are altered, since we may later be using this routine to generate a delay in a program that is holding important data in registers A, B, X, or Y at the time our delay subroutine is called. Therefore, if any of these registers are modified inside subroutine "wait_y_4us", their contents must be pushed onto the stack at the start of the subroutine, and then restored (pulled) from the stack, in the reverse order that they were stored, just before returning from this subroutine via the *return from subroutine* (RTS) instruction.

6.3 Verification of subroutine "wait_y_4us"

Test your subroutine by calling it using the test program shown in Figure 1. Note that this program calls wait_y_4us in order to generate a 50% duty cycle square wave on pin PM3 with a 20 ms period.

Figure 1. Test program that calls the "wait_y_4us" routine. (You must fill in the assembly code for the " wait_y_4us" routine in the space indicated.)

```
;*******************************************************************************
;   ECE331 Lab 3 Part 1
;   File: sqwave.asm (by KEH)
;    Square Wave Generating Program that calls the "wait_y_4us" subroutine
;    to generate a 50 Hz square wave on pin PM0 that is 10 ms high and 10 ms low.
;*******************************************************************************
                XDEF sqwave                   ; Export 'program entry point' symbol
                ABSENTRY sqwave               ; For absolute assembly: mark this as application entry point
                 INCLUDE 'mc9s12C128.inc'     ; Include 9S12C128 register symbol definitions.
                ORG $4000                     ;Start of Flash Program Memory on 9S12C128
sqwave:         lds  #$1000      ; Initialize program stack pointer to one location ABOVE the end of RAM
                                 ;(RAM on the 9S12C128 extends from $0400 to $0FFF.)
                movb #%1000,DDRM       ; Make PM3 an output.
                ldy  #2500             ; Note that 2500*4us = 10 ms delay time
next_cycle:     bset PTM, %1000        ; Set PTM3 high
                jsr wait_y_4us          ; Delay for 10 ms
                bclr PTM, %1000        ; Set PTM3 low
                jsr wait_y_4us         ; Delay for 10 ms
                bra next_cycle          ; go back to create next cycle of 50 Hz sq wave
;****** The "wait_y_4us" routine waits for "Y" multiplied by 4 us.  X,Y, D are preserved*****
wait_y_4us:
          ............... put your subroutine here.................
                 rts
                ORG      $FFFE
                fdb  sqwave            ; Initialize Reset Vector
```

A soft copy of the ***sqwave.asm*** "template file" shown in Figure 1 is available in the class folder. After you create the template project, copy the ***sqwave.asm*** file into this project folder. Then right click on the template ***main.asm*** file and select "***Remove***" to remove it from the project. Next right click on the ***Sources*** category in the project view and select ***Add Files***. Select ***sqwave.asm*** to add it to the project under the Sources category heading. You will need to fill in the "***wait_y_4us***" subroutine code in the place provided near the end of the file, following the hints of Section 6.2, before this project can be compiled. Note that the calling program must stay the same.

6.4 Lab Report Part 1

Run your program and verify its operation using your bench oscilloscope. Be sure to carefully measure the period of the square wave to verify proper timing. Include the adequately commented assembly source code of your "wait_y_4us" delay subroutine along with the test program above. Obtain the lab instructor's signature on your assembly listing.

7 Investigating the need for software switch debouncing: LED toggling program

Using the same hardware configuration from Lab 2 (Pushbutton SW on PM5 and LED on PM4, write a program that toggles, or changes the state of, the LED every time there is a ***rising edge*** on PM5.

7.1 Structure of LED toggling program

Your rising-edge sensitive LED toggling program should follow the following steps:

```
                Configure PM5 as input with no internal pullup (pushbutton SW)
                Configure PM4 as an output (LED);
next_pass:      Wait here until PM5 goes low (switch closed);
                Wait here until PM5 goes high (switch opened);
                Toggle PM4 (change state of LED);
                Go back to "next_pass";
```

Note that if the pushbutton switch does not bounce, each time the switch is pushed (generating a falling edge), nothing should happen to the LED, and each time the switch is released (generating a rising edge), the LED should change state, or toggle.

7.2 Running the LED toggling program without (and then with) contact debounce delay

Now open a new project, and write, compile, and run this program. When you run this program (say using LED1 and pushbutton PB1 on your project board), you should discover that LED1 does *not* toggle perfectly reliably (try 100 switch operations, and take note of how many toggling failures occur). Report this in your lab report. Furthermore, LED1 may sometimes change states not just when the pushbutton PB1 is released, but also when the pushbutton is pushed! As was discussed in Lab 1, this undesirable behavior is due to the fact that any mechanical switch contacts bounce for a few milliseconds, both when the button is pushed and also when the button is released. The LED will appear to change state if an odd number of rising edges are produced after each switch operation as the bounces die out. On the other hand, if an even number of rising edges is produced, the LED will appear to remain in the same state.

Actually, the pushbuttons on our project board are of rather high quality, and so the bounce time of PB1 is usually shorter than the instruction execution delays inherent in the toggling program, so the LED toggling should appear to be fairly reliable, though NOT perfectly reliable.

To observe less reliable toggling, connect a long hookup wire between the PB1 connection (there is a second connection to PB1 available on the IDE socket) on the project board, and then momentarily touch the other end of this wire to the ground "test lug" that is located at the upper left of the project board (marked GND1). *Keep the project board's pushbutton switch connected, as it provides the necessary pullup resistor!* Thus, we have replaced the high quality low-bounce pushbutton PB1 with a very low quality high-bounce wire "switch". As you move this wire on and off of the GND test lug contact, you should observe far less reliable toggling of the LED with this very bouncy switch. Once again, record the number of LED toggling failures out of 100 switch clicks.

Now we desire to make even the wire switch reliably toggle the LED. The solution is to call ***wait_y_4us*** both _after_ the step that waits for PM5 to go low and also _after_ the step that waits for PM5 to go high. (You will want to cut and paste the wait_y_4us subroutine from the previous project that generated the 100 Hz square wave.) Add these two subroutine calls, with Y set to 20000. Thus the delay subroutine will wait for 60000 * 4 µs = 240 ms, which is a very long switch debouncing time (about ¼ second), and this should be quite a bit longer than the worst-case bounce time of just about any pushbutton switch. You should now be able to observe reliable toggling of the LED even with our improvised wire switch (as long as you are careful to make clean contact between the wire and the test lug --- otherwise, this improvised switch could be made "infinitely bouncy")!

Now try changing Y to lower and lower values of delay. Find the shortest debounce delay (in milliseconds) for which your wire switch appears to work reliably. _Assume that "reliable" means 100% successful toggling 100 times in a row._ This delay corresponds to the worst-case bounce time of your wire switch. Now remove the wire and repeat this study using pusbutton PB1 on the project board.

7.3 Lab Report Part 2

Include a hard copy of your commented assembly source code of your (debounced) LED toggling test program. In your report be sure to indicate the shortest delay time that reliably debounces your wire switch. Also indicate the shortest delay time that reliably debounces PB1. Demonstrate both unreliable (with no delay) and reliable (with delay) toggling of the LED (using the improvised wire switch) to the lab instructor, and get his certifying signature on your assembly source code.

8 Debounced Keyswitch Array Input Subroutine "getkey"

The keypad we will use for this lab project is a 4x4 X-Y matrix 16-key keypad shown in Fig. 2. It can be checked out from the instrument room. At first glance, it might seem like the 16 SPST keyswitches would have to be connected to 16 I/O pins on the 9S12C128. However, because the keypad is arranged in a 4 X 4 "X-Y" matrix (4 rows and 4 columns), we only need to use 8 I/O pins, using a method called "keypad scanning" that is explained below.

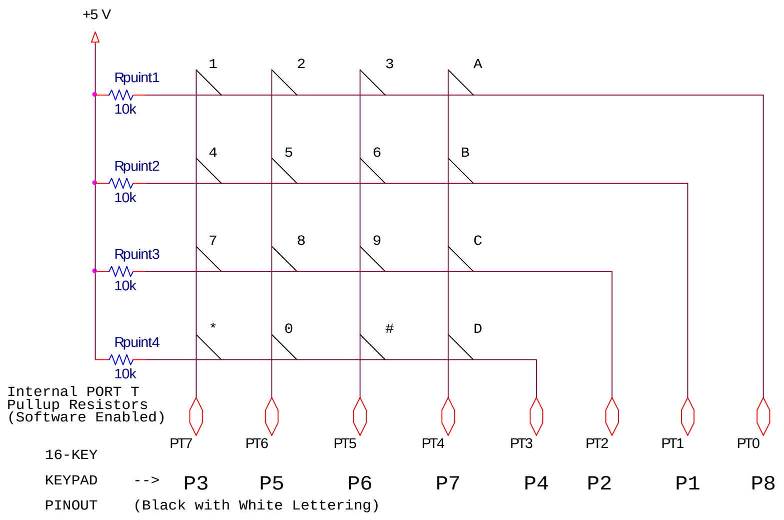
The interfacing circuit for the 4 X 4 keypad is shown in Fig. 2. Note that we have chosen to use the 8 Port T pins to interface to this keypad. When a keyswitch (represented by a short diagonal line in Figure 2) is depressed, it will close a contact between a column wire and a row wire.

To find out when a key is depressed (and also to identify the key's location in the keypad matrix), the four pins connected to the rows (PT3:0) are initially configured as inputs with internal pull-up resistors enabled, while the four pins connected to the columns (PT7:4) are initially configured as outputs. It is very important to enable internal (10 kΩ) pull-up resistors on the Port T pins that are configured as inputs. This is done by setting the "Port T Pull Device Enable" (PERT = $204) register to 0xFF. (See Section 3.3.1 and Figure 3-5 of the PIM 9C32 Block Guide, ***S12C128PIMV1.pdf***).

The output (column) pins (PT7:4) are then driven low, that is PT7:4 are set to %0000, and the input (row) pins (PT3:0) are repetitively read in a loop. While no key is yet depressed, all four of the input (row) pins should read high "1"; that is PT3:0 should be read as %1111. But if a key is depressed, the row pin that is connected to the pressed switch is driven low "0", since the

output resistance of a 9S12C128 output pin is much lower than the (internal) 10 k-Ohm pull-up resistor. Note that the row pin that reads low (all the rest should read high) corresponds to the row that the depressed key is in.

Figure 2. Interfacing a 4 X 4 keypad to the 9S12C128 microcontroller



In order to discover the column that the switch is in, the I/O directions of the column pins and row pins should next be reversed. This time all four of the output (row) pins (PT3:0) are driven low, that is PT3:0 = %0000. The column pins (PT7:4) may now be read by the M9S12C128 in order to find out which one of these pins is low (certainly one of them should be low, since a key was already found to be depressed). The column pin that reads low (the other 3 inputs should read high) corresponds to the column that the depressed key is in. The entire procedure for determining the row and column of a depressed key is illustrated in Figure 3. The drawing on the left shows the first step, where the columns (PT7:4) are driven low and the position of the pressed key is read as a "0"on the rows (PT3:0). The drawing on the right shows the second step, where the rows are driven low (PT3:0) are driven low, and the position of the pressed key is read as a "0" on the columns (PT7:4).

Note: It is important to check to see that Jumpers BZ and POT are disconnected (shorting jumpers moved to just one pin) on the Project Board. These are located right next to the Piezo Buzzer. Removing the BZ jumper should keep the Piezo Buzzer from clicking, and removing the POT jumper should allow one of the PORT T pull-up resistors to work properly.

Figure 3. Two-step scheme for determining the identity of single pressed key (X).

Step 1: Zeros driven onto columns, and rows are read. Position of "0" indicates which ***row*** the pressed key is in.

	0⇓	0⇓	0⇓	0⇓
1⇐				
0⇐		X		
1⇐				
1⇐				

Step 2: Zeros driven onto rows, and columns are read. Position of "0" indicates which ***column*** the pressed key is in.

	1⇑	0⇑	1⇑	1⇑
0⇒				
0⇒		X		
0⇒				
0⇒				

The 4-bit row identification pattern that was read from the bottom half of PORT T when the column lines (PT7:4) were driven to %0000, and the 4-bit column identification pattern that was read from the top half of PORT T when the column lines (PT3:0) were driven to %0000 may be "catenated" together into one 8-bit key identification code, or "keycode". (Notice that the upper 4 bits of this code should contain exactly one zero, and likewise, the lower 4 bits of this code should contain exactly one zero.) A "lookup table" can then be constructed that relates the keycode to the numerical value of the key ($0 - $F). An incomplete table is provided in Fig. 4 of this handout. Complete this table on your own. This table must be scanned by your program each time a key is pressed in order to learn the identity (value) of that key.

Figure 4. Keycodes for each key on the keypad

PT7	**PT6**	**PT5**	**PT4**	**PT3**	**PT2**	**PT1**	**PT0**	**DIGIT**
1	0	1	1	0	1	1	1	0
0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	0	2
								3
								4
								5
								6
								7
								8
								9
								A
								B
								C
								D
								*
								#

You should use the assembly directive "fcb" (form constant byte), or the equivalent "dc.b" (define constant byte) to construct a look-up table in flash ROM program memory (not RAM!) that will serve to convert an eight-bit keycode into the key value. The first part of this look-up table (based on the table of Fig. 4) appears below.

Keycode_Table: fcb $B7, $7E, $BE ……..

Note that the first entry in the Keycode_Table contains the keycode for the "0" key, the next entry is the keycode for the "1" key, etc. Thus, when a key is pressed on the keypad and a keycode has been determined by getkey, this table may be searched for a match with the keycode, and the position of the matching element becomes the key value ($0 - $F).

Using the same index offset at which the matching element that was found in the Keycode_Table, index this same distance into a companion ASCII_Table, the beginning of which appears below:

ASCII_Table: fcb '0', '1', '2',…….

This allows us to find the ASCII code of the key that was pressed.

When a key is found to be pressed, subroutine getkey should wait 40 ms for contact closure bounces to subside, and then wait for the key to be released. Once all keys are found to be released, the routine should delay for 40-ms before determining the key value from the keycode via a lookup table and returning from the subroutine, since keys can also bounce upon release.

To summarize the above discussion, here is an outline of the "getkey" subroutine:

```
START Getkey
Loop here until key is pressed;
Delay 40 ms to wait for switch contact bouncing to die out;
Get a 4-bit row position code;
Get a 4-bit column position code;
Catenate the two 4-bit codes together into a "keycode".
Find which key has been pressed using the Keycode_Table
        lookup table
Determine the ASCII code of the key that was pressed
        using the ASCII_Table lookup table.
Loop here until key is released.
Delay 40 ms to wait for switch release bouncing to die out.
Return with the ASCII code of the key in Acc B;
END Getkey
```

8.1 Waiting for Port T changes to settle

Due to the high speed at which the 9S12C128 executes instructions, it is necessary to insert a short delay (I suggest that you call the "wait_y_4us" routine with Y = 25 or so) between when the zeros are driven out on either the high or the low half of Port T and when the other half of Port T is read. This delay is necessary because it takes time for the voltage on the input pins that are being read stabilize after a change occurs at the output pins of Port T. This is because there is a significant rise time (perhaps several microseconds) that must be waited out on an input pin due to the RC time constant formed between the internal 10 kΩ pull-up resistance and the parasitic capacitance exhibited by the keypad. If you forget to put this short "settling delay" into your getkey subroutine, you may find that your getkey subroutine works fine when you are single-stepping through it, but that it works unreliably when the program is run at full speed!

8.2 Lab Report Part 3

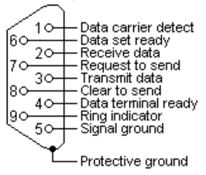
Demonstrate a short test program that calls subroutine getkey to verify its reliable operation. Include a commented assembly listing of your program in your lab report.

9 Write an asynchronous serial output routine "outchar"

Write a subroutine "outchar" as yet another new project that will send an ASCII code in Accumulator B to the 9S12C128's serial communication terminal's display through the DB9 RS232C interface. This subroutine should be called by a short test program that calls the subroutine to print several ASCII characters over and over again. This subroutine will use the serial communication interface (SCI) functional block of the 9S12C128 microcontroller to implement RS232 asynchronous serial data communication. (See the "HCS12 Serial Communication Interface Block Guide", ***S12SCIV2.pdf***.)
The "receive data" input pin PS0 (RxD) is used to receive serial asynchronous data from an RS232 serial communications terminal.. The "transmit data" output pin PS1 (TxD) is used to transmit serial asynchronous data to a terminal. The 9S12C128 microcontroller's two serial communication pins (TxD and RxD) are connected to an RS232 terminal through an on-module interface chip (Max3232CD) that converts the 0 V / 5 V CMOS logic voltage levels to the corresponding +10 V / -10 V RS232 voltage levels. These RS232-compatible signals are connected to Pins 2 and 3 on the 9-pin D-shell serial port connector that resides on the 9S12C128 module. The wiring of this connector is shown in Figure 6. We are using only the RxD, TxD, and Signal Ground pins in our application, since we will NOT use "Hardware Flow Control".

Once the Hyperterminal session is started, if you unplug the serial cable from the module, and carefully short Pins 2 (RxD) and 3 (TxD), without touching the metal grounding shield, using a key or a screwdriver, you should be able to perform a "loopback test" on the terminal; that is when you depress a key on the keyboard, the transmitted RS232 signal from the keyboard TxD is routed back into the RxD input to the display, and the corresponding character should be displayed on the monitor. This test gives verifies that the serial cable (and Hyperterminal) is operating properly.

Figure 6. RS232 DB9 Pinout (Front, Pin-Side View for Female Connector)



Please study the SCI Block Guide to make sure that you understand why the following registers must be initialized at the beginning of the outchar subroutine in the following way:

SCIBDL = 13	; SCI Baud Rate = 2 MHz / (16 * SCIBDL)
SCIBDH = 0	; Note: 9600 Baud = 2 MHz / (16 * 13)
SCICR1 = $00	; 1 start bit, 8-bit data, 1 stop bit, no parity
SCICR2 = $0C	; transmit and receive enabled, no interrupt

An ASCII character (byte) is sent to the terminal when the data register SCIDRL is written with that byte. But first your subroutine must wait (loop) until the "Transmit Data Register Empty" (TDRE) bit, which is Bit #7 in the SCISR1 register, is a "1", indicating that the SCI transmit data register is indeed empty, and thus ready to receive new data before the new data byte (in Accumulator B) can be written to the SCIDRL. The SCI hardware will then automatically begin to shift and transmit the contents of SCIDRL out of the TxD pin to the terminal's display at the previously configured 9600 bit per second "baud rate".

In order to observe the proper operation of your "outchar" routine, you will have to start a "serial communications terminal" program running on your PC, such as Hyperterminal, which comes with Windows. You will initially have to set up Hyperterminal by following these steps:

1. Click on ***Start – Programs – Accessories – Communications – Hyperterminal***
2. Enter a connection name such as "***ECE331***_term". Hit ***OK***
3. In the "Connect To" window, select "Connect Using" ***COM1*** (or COM2, as needed). Click ***OK***.
4. In the properties window, change Bits Per Second: ***9600***, Data Bits***: 8***, Parity: ***None***, Stop Bits: ***1***, Flow Control: ***None***. Click ***OK***.
5. Click on ***File – Save As***, and save your Hyperterminal session to the "Windows Desktop", so you may click on the resulting desktop icon, rather than having to go

through all of these setup steps the next time you want to re-open your Hyperterminal communication window.

6. The Hyperterminal communication window should now be visible.

9.1 Lab Report Part 4

Write and demonstrate a short test program that calls the "outchar" subroutine, and have it print several ASCII characters over and over again in an endless loop. First carefully and gently connect the serial port cable that came with your CSMB12C128 module between your PC's serial port (or one of the lab PC's serial ports if your laptop PC does not have a serial port) and your 9-pin D-shell serial port connector that is located on the CSMB12C128 module (NOT the one located on the project board). The desired character stream should be displayed in the Hyperterminal window when you run this program using the BDM debugger.

Include a commented assembly listing of your "outchar" subroutine and its associated test program. Also include a screen shot of your Hyperterminal window, showing that the desired characters have been displayed.

10 Putting it all together: final system integration!

Now create another new project that uses the short program of Figure 7 to call the various subroutines developed above to implement a simple hexadecimal keypad scanning system, where any sequence of keys (0,1, ...,E,F) can be typed on the keypad, with the corresponding sequence of hexadecimal numbers subsequently displayed on the PC's Hyperterminal serial communication screen.. The following separately tested software "module" subroutines from the previous portions of this lab should be cut and pasted into the program file after the calling program: wait_y_4us, getkey, and outchar.

Figure 7. Final system integration program

```
Entry:      LDS #$1000          ;Init Stack Ptr near top of RAM just below UBUG Stack
nextchar:   JSR getkey
            JSR outchar
            LDAB #$0A           ;send a line feed to terminal
            JSR outchar
            LDAB #$0D           ;send a carriage return
            JSR outchar
            BRA nextchar
            ;……………………………………………………………..
            ;Subroutines wait_y_4us, getkey,
            ;and outchar are all put here
            ;……………………………………………………………..
```

10.1 Lab Report Part 5

Your memo-style report must include (1) a commented assembly source file of your "wait_y_4us" delay subroutine along with the test program this final program. This listing must be signed by the instructor. (2) A hard copy of your commented assembly source code of your (debounced) LED toggling test program. Be sure to indicate the shortest delay time that you discovered that will reliably debounce your particular SPST pushbutton switch that was made from a wire that was touched to ground, as well as the shortest delay time that reliably debounces

the built-in pushbutton PB1 on the project board. This assembly program must be signed by the instructor. (3) Hard copy of your commented test program that calls subroutine getkey to verify its reliable operation. (4) Commented assembly source file of your "outchar" subroutine and its associated test program. (5) Include a listing of the final "system integration" program that scans the 4x4 matrix keypad and provides ASCII RS232 serial output. This file must be signed by the instructor to verify that it was successfully demonstrated.

11 Appendix A: ASCII Table

	0	1	2	3	4	5	6	7
0	NUL	DLE	Space	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	}
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	delete