ECE 4510 Introduction to Microprocessors

Software Review

Dr. Bradley J. Bazuin
Associate Professor
Department of Electrical and Computer Engineering
College of Engineering and Applied Sciences

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
Modern Computers

• They are everywhere and in just about everything.
  – Ubiquitous computing
  – Cloud computing

• We are all users …
  – Some of us are knowledgeable users
  – Fewer understand basic “computer architecture”
  – Fewer yet are programmers
  – And a very few are designers, architecting and building the next generation.
The Computing Problem


Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
Software Development

- System Engineering
  - Define system architecture and architectural elements
  - Partition system for engineering elements (e.g. antenna, RF, digital/programmable logic, DSP, general CPU, real-time software, command & control software with GUI

- Software Development (DSP and/or CPU)
  - Design
  - Code
  - Test
  - SW-SW Integration
  - SW-HW Integration
  - System Integration
  - Acceptance Testing
Software Development Issues

• Software development starts with problem definition
  – Problem presented by the application must be fully understood before any program can be written

• Next step is to lay out an overall plan of how to solve the problem
  – The plan is also called an algorithm
  – Algorithm is a sequence of computational steps that transforms the input into the output
  – An algorithm can be expressed in pseudo code that is very much like C or Pascal (or the target language for the code)
  – An algorithm provides not only the overall plan for solving the problem but also documentation for the S/W to be developed
The CPU and Peripherals: Tools Available for Software

- On Chip Memory
  - Random Access Memory (RAM-14K)
  - Electrically Erasable Memory (EETS-4K)
  - Flash Programmable Memory (FTS-512K)
  - SFFFF (high add.)

- Background Debug Module (BDM)
- Breakpoint Module (BKP)
- Interrupt Module (INT)

- Central Processing Unit (CPU)

- Parallel Port Integration Module (PIM)

- Analog to Digital Converter (ATD – 0 and 1)

- Pulse Width Modulation (PWM)

- Clock and Reset Generator (CRG)

- Serial Peripheral Interface (SPI 0, 1, and 2)

- Controller Area Network (CAN – 0 to 4)

- Enhanced Capture Timer (ECT)

- Serial Communications Interface (UART/SCI-0 and 1)

- Inter-IC Bus (IIC)

- Other Automotive Standards
  - ByteFlight (BF) and J1850 (BDLC)

---

Software Development Environment

- Create a project in the environment
- Load files containing code into the project
- For high-level languages: Compile into assembly language
- For assembly language: Assemble into object code module
- Link multiple object code modules into machine code module
- Load machine code module onto the target host
- Executing machine code
  - For development: Use a monitor program or debugger to run machine code
  - For users: Execute machine resident code
High-level Language Preferred

• Syntax of a high-level language is “similar to English” (?!)
• A translator is required to translate the program written in a high-level language -- done by a compiler.
  – There are two types of compilers: native compiler and cross compiler.
• High-level languages allow the user to work on the program logic at higher level and achieve higher productivity.
• Source code
  – A program written in assembly or high-level language
• Object code
  – The output of an assembler or compiler
Your ECE4510/ECE5530 Development Environment

- Software Development Environment
  - The Imagecraft Freescale CPU12 Development Tools
  - ICCV7 for CPU12 – refereed to as ICC12 around WMU
    - http://www.imagecraft.com/

- CPU Embedded Debugging Tool
  - Freescale D-Bug12

- NoICE with the background debugging module (BDM)
  - ICE refers to an “In-Circuit Emulator”
Code Flow Loop Diagrams

(a) For $I = i_1$ to $i_2$ DO $S$

(b) For $I = i_2$ downto $i_1$ DO $S$

Figure 2.4 An infinite loop

Figure 2.5 For looping construct

Figure 2.6 The While ... Do looping construct

Figure 2.7 The Repeat ... Until looping construct

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
Subroutine Hierarchy

- Structure of a modular program can be visualized as shown
- Principles of program design involved in high-level languages can be applied to assembly language programs
  - Subroutine “objects” with calls and returns
Interrupts For Embedded Processing

• Initialize the interrupting function and set the local interrupt enable bit
  – clear pending interrupts if possible
• Make sure a valid interrupt service routine (ISR) exists
• Make sure the system can find the ISR code
  – A ISR address table is often used for defined types of interrupts
  – Also called an interrupt vector (IV) table (address of address)

• When ready, set the global interrupt enable
  – Pending interrupts will immediately be serviced
  – Interrupt ISR will start where the IV says, even if you forgot to define where in the IV
General Real-Time Code Flow

- Sequential Code with Infinite Loop
  - Process as time is available
- Interrupts that must be immediately processed

Modified for Sleep (Low Power)

- Sequential Code with Infinite Loop
  - Process followed by sleep mode
  - Sleep mode exited by an interrupt
  - Only “awake” when processing is needed!
Software Operating States

A: Initialization
B: time in seconds data entry
C: accept time, idle until turned on
D: turn on and start counting down the time
E: undefined
F: emergency stop

“User proofing" requires additional state transitions
Sunseeker Driver Controller
“State” Update Code (1 of 2)

```c
// Update the DC Mode
switch(dcMODE){
    case POWERUP:
        if(switches_new & SW_IGN_ON){
            enable = FALSE;
            switches_new &= ~(SW_IGN_ON);
        } else {
            dcMODE = PRECHARGE;
        } break;
    case PRECHARGE:
        if(switches_new & SW_IGN_ON){
            enable = FALSE;
            switches_new &= ~(SW_IGN_ON);
            switches_out_new |= 0xFF00;
            if(bps_precharge_done) dcMODE = DISABLE;
        } break;
    case DISABLE:
        if(enable){
            if(reverse) dcMODE = REV_RDY;
            else dcMODE = FWD_RDY;
        } switches_new &= ~(SW_REGEN);
        break;
    case FWD_RDY:
        if(!enable){
            dcMODE = DISABLE;
        } else {
            if(reverse) dcMODE = REV_RDY;
            else if(adcvalue1 > ADC_MIN || moving) dcMODE = FWD_DRV;
        } switches_new &= ~(SW_REGEN);
        break;
    }
}
```

dcMode should be named “dcState” for the “operating state” the car is in

ECE 4510/5530

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
Sunseeker Driver Controller
“State” Update Code (2 of 2)

```c
enum MODE {
    POWERUP,
    PRECHARGE,
    DISABLE,
    FWD_RDY, FWD_DRV,
    REV_RDY, REV_DRV,
    REGENEN,
    CRCNTRL
} dcMODE;
```

case FWD_DRV:
    if(!enable) { dcMODE = DISABLE; }
    else if(!moving &amp;&amp; brake) { dcMODE = FWD_RDY; }
    else if(cruise){
        dcMODE = CRCNTRL;
        switches_new &= ~(SW_REGEN);
        cruise_velocity = actual_velocity;
        cruise_current = avg_set_current;
        cruise_steps = 0;
    }
    else if(regen){
        dcMODE = REGENEN;
    }
    break;

case REV_RDY:
    if(!enable) { dcMODE = DISABLE; }
    else {
        if(reverse) { dcMODE = FWD_RDY; }
        else if(adcvalue1 > ADC_MIN || moving) { dcMODE = REV_DRV; }
        switches_new &= ~(SW_REGEN);
        break;
    }

case REV_DRV:
    if(!enable) { dcMODE = DISABLE; }
    else if(!moving &amp;&amp; brake) { dcMODE = REV_RDY; }
    switches_new &= ~(SW_REGEN);
    break;

case REGENEN:
    if(!enable) { dcMODE = DISABLE; }
    else if(!regen) { dcMODE = FWD_DRV; }
    switches_new &= ~(SW_REVERSE);
    break;

case CRCNTRL:
    if(!enable) { dcMODE = DISABLE; }
    else if(!cruise || brake) { dcMODE = FWD_DRV; }
    switches_new &= ~(SW_REVERSE | SW_REGEN);
    break;
```

Using the previous example

- You need one switch-case statement to update the current state to the next state (or not change)
- A second switch-case block may be needed to perform operations based on the state.

- The two switch-case blocks could be merged into one to “minimize code”, but it may not be as clear to the person reading/debugging the code as to what is going on.
  - two processes:
    (1) what to do in a state and
    (2) what is the next state
Alternate State Method 1

```
asm("sei"); // disable interrupts
// initialize everything here
state = "A";
asm("cli"); // enable interrupts

while(1) {
    while(state == "A") {
    }

    while(state == "B") {
    }

    while(state == "C") {
    }

    ... etc ...
}
```

- Exist in a state until a condition changes the “state” variable.
  - locked in a state
  - primary while loop does not repeat regularly
  - must have a means to modify “state” either in the state or by an interrupt with “state” as a global variable

Alternate State Method 2

- Chained if … else if … else
  - primary while loop repeats regularly
  - not locked in a state
  - state update or modification can be performed separately

```c
asm("sei");  // disable interrupts
// initialize everything here
state = "A";
asm("cli");  // enable interrupts

while(1)
{
    if(state == "A")
    {
        ...
    }
    else if(state == "B")
    {
        ...
    }
    else
    {
        ...
    }

    ... etc ...
}
```
Clock and Reset Generation Block (CRG) (CRG_SW code setup)
Set Choice of Clock Sources

![Diagram of clock generation circuit]

- Oscillator (OSCCLK)
- PLL clock (PLLCLK)
- Clock monitor
- Phase lock loop
- PLLSEL or SCM
- Sysclk
- Core clock
- E-Clock
- Real Time Interrupt
- Comp. Operating Properly

Gating condition

- Extal
- Xtal

Figure 6.15: HCS12 clock generation circuit.
Phase Lock Loops

- A Phase Lock Loop is capable of providing an integer change up or down in the clock rate input to the PLL
  - A voltage controlled oscillator is used
CRG: Clock and PLL Setting

void pll_init(void)
{
    #ifdef __SET_PLL
    // PLLCLK = 2 * OSCCLK * (SYNR+1)/(REFDV+1)
    // ECLK = PLLCLK/2
    CLKSEL &= ~PLLSEL;
    PLLCTL |= PLLON | AUTO | ACQ;
    REFDV = (XTALRATE/PLLCOMPRATE) - 1;  // from 1 to 16
    SYNCR = (BUSRATE/PLLCOMPRATE) - 1;  // from 1 to 64
    asm("nop");  // Allow time for the PLL to start and settle
    asm("nop");
    asm("nop");
    asm("nop");
    while((CRGFLG & LOCK) == 0x00)
    {
        asm("nop");
    }
    CLKSEL |= PLLSEL;  // use the PLL as the system clock
    PEAR &=-NECLK;  // PortE pin 4 is E-clk output
    MODE &=-0xEF;  // Set to special single chip operation
    #endif
    #ifdef __ECLK_ACCESS
    PEAR &=-NECLK;  // PortE pin 4 is E-clk output
    MODE &=-0xEF;  // Set to special single chip operation
    #endif
}

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
RTI Counter Chain

Note: OSCCLK not SYSCLK and not E-Clock

Figure 4-6 Clock Chain for RTI
CRG: RTI Initialization and ISR

```c
void crg_init(void)
{
    // Port P pin 7 is the Adapt Board LED
    DDRP |= DDRP7;
    PTP |= PTP7;
    // RTI Initialization
    RTICTL = 0x27;  // 8 x 2^11 division OSCCLK = 16 MHz -> 1.024 msec
    CRGINT |= RTIE;  // enable the rti interrupt
}

#pragma interrupt_handler rti_isr
void rti_isr(void)
{
    static unsigned int count = 0x0000;

    CRGFLG |= RTIF;  // Reset the interrupt flag
    count = count +1;
    // 500 x 1.024 msec = 0.512 sec
    if(count == 500)
    {
        PTP ^= PTP7;
        count = 0;
    }
}
```

```c
#ifdef __FLASH_LOAD
#pragma abs_address:vrti // Initialize the Interrupt
Vector address
void (*interrupt_vectors[]) (void) = {rti_isr};  // Assign the function pointer
#pragma end_abs_address // to the ISR
#endif

#else
#pragma abs_address:0x3E70 // Initialize the Interrupt
void (*interrupt_vectors[]) (void) = {rti_isr};  // Assign the function pointer
#pragma end_abs_address // to the ISR
#endif
```
Parallel Pin Init:
SPI Port Init (for SS Pin)

```c
void init_spi0(void)
{
    // SPI0 port pin name definitions (not needed, but nice)
    #define MISO0 PT4;
    #define MOSI0 PTS5;
    #define SCK0 PTS6;
    #define SS0n PTS7;
    // SPI0 port interface macros
    #define spi0_select PTS &= ~PTS7;
    #define spi0_deselect PTS |= PTS7;
    SPI0BR = 0x20; // set baud rate to 24 MHz/6 = 4 MHz
    SPI0CR1 |= SPE | MSTR | CPHA; // Enable, Master, SCLK high active low, even phases
    SPI0CR1 &= ~(SSOE | CPOL ); // do not use the SSn output pin
    SPI0CR2 |= SPISWA1; // stop sclk in wait mode
    SPI0CR2 &= ~(MODFEN | SPC0); // do not use the SSn output pin, normal MISO MOSI pins
    DDRS |= PTS7; // SS0n parallel output pin
    PERS |= PERS4; // enable pullup/down on SPI0 MISO pin
    PERS &= ~(PERS5 | PERS6 | PERS7); // disable pullup/down on SPI0 pins
    PPSS &= ~(PPSS4); // pull-ups on SCI0 MISO
    WOMS |= WOMS4; // open-drain drive on SCI0 MISO
}
```
Class Keypads
A GetKey function

```c
#define keypad PORTA
#define keypad_dir DDRA

char getkey(void)
{
    char key_array[4][4] = {{0, 7, 4, 1}, {15, 8, 5, 2}, {14, 9, 6, 3}, {13, 12, 11, 10}};
    char rmask, cmask, row, col;

    cmask = 0xEF; // init keypad scan
    for (col = 0; col < 4; col++) //
    {
        rmask = 0x01; // Test the 0th row
        keypad = cmask;
        for (row = 0; row < 4; row++)
        {
            if (!(keypad & rmask)) // key switch detected pressed
            {
                keypad = 0xFF; // standby keypad values
                return (key_array[col][row]);
            }
            rmask = rmask << 1;
        }
    }
    cmask = (cmask << 1) | 0x0F; // sequence of 0xEF, 0xDF, 0xBF and 0x7F
    keypad = 0xFF; // standby keypad values
    return (0xFF);
}
```
Main Code: Periodic Keypad Test

```c
while(1)
{
    // Do something here
    asm("nop");

    // Key scanning designed for every while loop
    // Looking for the key to be pressed and released
    //
    if(check_keypad_flag)
    {
        current_key = getkey(); // retrieve a key value

        if(last_key <> 0xFF) {
            key_count++; // count the key press loops
            key_status = 0x01; // status is 0x01 invalid press
            if(key_count>KEY_THRESH) {
                key_status = 0x02; // status is 0x02 valid press
                if(current_key == 0xFF) {
                    key_status = 0x03; // status is 0x03 saved value
                    key_ptr++ = last_key;
                }
            } else {
                key_count = 0;
                key_status = 0x00;
            }
        } else {
            last_key = current_key;
        }
    }
}
```

If key_status=0x03, do something about the value written!
# define keypad PORTA
# define keypad_dir DDRA
# define KEY_THRESH1000

void main(void)
{
    int ii;
    char check_keypad_flag;
    char key_status, last_key, current_key;
    char key_input[10];
    char *key_ptr;
    int key_count;

    COPCTL=0x00;
    asm("sei");

    // Initialize keypad and key variables
    keypad_init(); // set up port pins
    last_key = 0xFF;
    key_count = 0;
    key_status = 0x00;
    key_ptr = &key_input[0];
    for(ii=0; ii<10; ii++) // initialize key_input
        {
            key_input[ii]=0;
        }

    asm("cli");
5x7 Matrix with Source and Sink ICs

MIC5891 Source
LTP-757G or similar Display
ULN2803A Sink
[note shown 74HCT595 shift reg.]

(Future Sink: TPIC6C596N)
Operating the MIC5891YN

A. Minimum data active time before clock pulse (data set-up time) 75ns
B. Minimum data active time after clock pulse (data hold time) 75ns
C. Minimum data pulse width .150ns
D. Minimum clock pulse width 150ns
E. Minimum time between clock activation and strobe 300ns
F. Minimum strobe pulse width 100ns
G. Typical time between strobe activation and output transition 1.0μs
Bit Banging MIC row data

- Using software and individual bit-level ports to create a more complex signaling stream (PTP.0 data, PTP.1 clock)

```c
row_load(rows_excited[jj]);
void row_load(char temp){
    int ii;
    for (ii=0;ii<8;ii++) {
        bit_value = temp & 0x01; // Determine value
        if(bit_value == 0x01) PTP |= 0x01; // Data output
        else PTP &= ~(0x01);
        asm("nop"); // setup > 75ns
        asm("nop");
        PTP |= 0x02; // clock high
        asm("nop"); // pulse > 150 ns
        asm("nop");
        asm("nop");
        PTP &= ~(0x02); // clock low
        temp = temp>>1;
    }
}
```
Logic for Selecting Columns

- 74HCT595: 8-bit serial-in, serial or parallel-out shift register with output latches; 3-state
### Functional Table

#### Table 3. Function table

<table>
<thead>
<tr>
<th>Control</th>
<th>Input</th>
<th>Output</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>a LOW-level on MR only affects the shift registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>empty shift register loaded into storage register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>shift register clear; parallel outputs in high-impedance OFF-state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logic HIGH-level shifted into shift register stage 0. Contents of all shift register stages shifted through, e.g. previous state of stage 6 (internal QnS) appears on the serial output (Q7S).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>contents of shift register stages (internal QnS) are transferred to the storage register and parallel output stages</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>contents of shift register shifted through; previous contents of the shift register is transferred to the storage register and the parallel output stages</td>
</tr>
</tbody>
</table>

- **fmax**: 20 MHz max
- **clock high**: 24 ns min
- **set-up**: 24 ns min
- **hold**: 3 ns min
- **prop**: 63 ns max

---

*Introduction to Software & Hardware*  
Thomson Delmar Learning, 2006.
Bit Banging column shift register

- Using two cascaded 74HCT595

```c
int jj;
PTP |= 0x10;  \ Set the one into the shift register
PTP |= 0x20;  \ shift register clock rise
PTP &= ~(0x30);  \ shift register clock fall
for (jj=0; jj<15; jj++) {
    row_load(rows_excited[jj+offset]);  \ load the row values
    PTP |= 0x20;  \ shift register clock rise
    PTP &=(~(0x80));  \ Output enable row and column
    PTP &= ~(0x20);  \ shift register clock fall
    delay(how long is the LED on)  \ one row of LEDs is on
    PTP |= 0x80;  \ Output disable row and column
    }  \ next row/column
```
if(display5x7_advance_flag)
{
  display5x7_advance_flag = FALSE;
  PORTA |= 0x08; // Output disable row source
  switch(display_count)
  {
    case 0:
      PORTA |= 0x10; // Set one into the first shift register
      PORTA |= 0x20; // shift register clock rise
      PORTA &= ~(0x30); // shift register clock fall, remove input data
      display_count++; // increment counter
      break;
      row_location = display_count + offset - 1; // row index
      row_load(row_data_array[row_location]); // load the row value
      PORTA |= 0x20; // shift register clock rise
      PORTA &= ~(0x08); // Output enable row source
      PORTA &= ~(0x20); // shift register clock fall
      display_count++; // increment counter
      break;
    default:
      PORTA |= 0x20; // shift register clock rise
      PORTA &= ~(0x20); // shift register clock fall
      display_count = 0;
      break;
  }
} // end display5x7_advance

// 1 cycle load column bit
// 15 cycles load row data shift column bit and enable
// 1 cycle shift column bit to 16th/empty column
// Operational rate approx. 60 Hz * 17 ~ 1020 Hz
// Using rti @ 1.024 msec => 976 Hz
// or 57.4 complete scans/sec

Optical Rotary Shaft Encoder

- Detect turn direction by which rising edge leads.
  - 16 or 32 position
  - 11.25 degree or 22.5 degree for code change
- Similar device used on Sunseeker to engage and control cruise control
Sunseeker Shaft Tracking Code

// Keep track of encoders
enc1_new = (P1IN & (ENC1_B | ENC1_A));
switch(enc1_old){
    case STATE_1_A: // 00
        if( enc1_new == STATE_1_B ){
            encoder1--;
            cruise_velocity -= CRUISE_STEP;
        }
        else if( enc1_new == STATE_1_D ){
            encoder1++;
            cruise_velocity += CRUISE_STEP;
        }
        break;
    case STATE_1_B: // 01
        if( enc1_new == STATE_1_C ){
            encoder1--;
            cruise_velocity -= CRUISE_STEP;
        }
        else if( enc1_new == STATE_1_A ){
            encoder1++;
            cruise_velocity += CRUISE_STEP;
        }
        break;
    case STATE_1_C: // 11
        if( enc1_new == STATE_1_D ){
            encoder1--;
            cruise_velocity -= CRUISE_STEP;
        }
        else if( enc1_new == STATE_1_B ){
            encoder1++;
            cruise_velocity += CRUISE_STEP;
        }
        break;
    case STATE_1_D: // 10
        if( enc1_new == STATE_1_A ){
            encoder1--;
            cruise_velocity -= CRUISE_STEP;
        }
        else if( enc1_new == STATE_1_C ){
            encoder1++;
            cruise_velocity += CRUISE_STEP;
        }
        break;
    default:
        break;
}
enc1_old = enc1_new;
The HCS12 Timer Elements

- 16-bit free-running main timer
  - Prescaler
- 16-bit modulus downcounter
  - Prescaler
  - Load
- Control Registers
- Interrupt Registers
- Capture/Compare Registers

Figure 1-1 Timer Block Diagram

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
Delay Using ECT Down Counter

```c
void delayby1ms(int k)
{
    int i;
    TSCR1 |= TFFCA; /* enable fast timer flag clear */
    for (i = 0; i < k; i++) {
        MCCTL = 0x07; /* enable modulus down counter with 1:16 as prescaler */
        MCCNT = 1500; /* let modulus down counter count down from 1500 */
        while(!(MCFLG & MCZF));
        MCCTL &= ~0x04; /* disable modulus down counter */
    }
}
```

See Textbook CD: Utilities/delay.c

The above is from the current textbook CD
Current Textbook Delay Code

- **Delay.c**
  - `void delayby10us(int k);`  // prescale 1, MCCNT = 240
  - `void delayby50us(int k)`  // prescale 1, MCCNT = 1200
  - `void delayby1ms(int k);`  // prescale 16, MCCNT = 1500
  - `void delayby10ms(int k);`  // prescale 16, MCCNT = 15000
  - `void delayby100ms(int k);`  // prescale 16, loop 10 times with MCCNT = 15000

These are “blocking” delay calls.
Interrupts may occur, but nothing else is happening in the CPU.
/* time delay computation is based on */
/* 24 MHz crystal oscillator. */

void delayby1ms(int k)
{
    int ix;
    TSCR1 = 0x90; /* enable TCNT and fast timer flag clear */
    TSCR2 = 0x06; /* disable timer interrupt, set prescaler to 64 */
    TIOS  |= BIT0; /* enable OC0 */
    TC0 = TCNT + 375;
    for(ix = 0; ix < k; ix++) {
        while(!(TFLG1 & BIT0));
        TC0 += 375;
    }
    TIOS  &= (~BIT0); /* disable OC0 */
}

See Textbook CD:
Utilities/delay.c
The above is from the old textbook CD
Periodic Event Timing
Possible Code?

```c
void modcnt_init(void)
{
    // Modulus Down Counter Initialisation
    TSCR1 |= TFFCA; /* enable fast flag clear */
    /* interrupt enable, modulus mode, prescale 16*/
    MCCTL = MCZI | MODMC | MCPR1 | MCPR0;
    MCCNT = 1500; /* 1 msec interrupts */
    MCCTL |= MCEN; /* start counting */
}

#pragma interrupt_handler modcnt_isr
void modcnt_isr(void)
{
    static unsigned int flag_count = FLAG_TIME;
    extern unsigned char flag1;
    MCFLG |= MCZF; // Reset the interrupt flag
    // Trigger flag events (indicators)
    flag_count --;
    if( flag_count == 0 ){
        flag_count = FLAG_TIME;
        flag1 = TRUE;
    }
}

#pragma abs_address:vtimmdcu // Initialize the Interrupt Vector address
void (*interrupt_vectors[]) (void) = {modcnt_isr}; // Assign the function pointer
#pragma end_abs_address // to the ISR.

FLAG_TIME must be defined in a header file
```
# define FLAG_TIME 500 // 0.5 sec interrupts
unsigned char flag1 = FALSE; // assign a global flag
int main(void)
{
    asm("cli"); // enable the modulus down counter
    modcnt_init(); // start interrupts

    while(1) {
        asm("nop"); // every loop code goes here

        if(flag1){
            flag1 = FALSE; // Reset the flag
            asm("nop"); // do something every 0.5 sec
        }
    }
    return 0;
}

A different flag can be created for every periodic operation required (i.e. keypad, 5x7 matrix, etc.)
# Example 8.2: C Program for Period Measurement

```c
#include "c:\egnu091\include\hcs12.h"
void main(void) {
    unsigned int edge1, period;
    TSCR1 = 0x90; /* enable timer counter, enable fast flag clear*/
    TIOS &= ~IOS0; /* enable input-capture 0 */
    TSCR2 = 0x06; /* disable TCNT overflow interrupt, set prescaler to 64 */
    TCTL4 = 0x01; /* capture the rising edge of the PT0 pin */
    TFLG1 = C0F; /* clear the C0F flag */
    while (!(TFLG1 & C0F)); /* wait for the arrival of the first rising edge */
    edge1 = TC0; /* save the first captured edge and clear C0F flag */
    while (!(TFLG1 & C0F)); /* wait for the arrival of the second rising edge */
    period = TC0 - edge1;
    asm ("swi");
}
Example 8.3: C Program for Pulse Width Measurement (1 of 2)

```c
#include <hcs12.h>
#include <vectors12.h>

unsigned diff, edge1, overflow;
unsigned long pulse_width;

void INTERRUPT tovisr(void);
void main(void)
{
    COPCTL=0x00;
    asm("sei");
    overflow = 0;
    TSCR1 = 0x90; /* enable timer and fast flag clear */
    TSCR2 = 0x05; /* set prescaler to 32, no timer overflow interrupt */
    TIOS &= ~IOS0; /* select input-capture 0 */
    TCTL4 = 0x01; /* prepare to capture the rising edge */
    TFLG1 = C0F; /* clear C0F flag */
    while(!(TFLG1 & C0F)); /* wait for the arrival of the rising edge */
    TFLG2 = TOF; /* clear TOF flag */
    TSCR2 |= 0x80; /* enable TCNT overflow interrupt */
    asm("cli");

    Example 8.3: C Program for Pulse Width Measurement (1 of 2)
```

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
Example 8.3: C Code Pulse Width (2 of 2)

```c
edge1 = TC0; /* save the first edge */
TCTL4 = 0x02; /* prepare to capture the falling edge */

while (!(TFLG1 & C0F)); /* wait for the arrival of the falling edge */

diff = TC0 - edge1;
if (TC0 < edge1)
    overflow -= 1;
pulse_width = overflow * 65536u + diff;

asm ("swi");
}

void INTERRUPT tovisr(void)
{
    TFLG2 = TOF; /* clear TOF flag */
    overflow = overflow + 1;
}
```
Example 8.4 Output Waveform

```c
#include "c:\egnu091\include\hcs12.h"
#define hi_time 900
#define lo_time 2100
void main (void)
{
    TSCR1 = 0x90;    /* enable TCNT and fast timer flag clear */
    TIOS  |= OC0;   /* enable OC0 function */
    TSCR2 = 0x03;    /* disable TCNT interrupt, set prescaler to 8 */
    TCTL2 = 0x03;    /* set OC0 action to be pull high */
    TC0    = TCNT + lo_time;       /* start an OC0 operation */
    while(1) {
        while(!(TFLG1 & C0F)); /* wait for PT0 to go high */
        TCTL2 = 0x02;    /* set OC0 pin action to pull low */
        TC0   += hi_time;    /* start a new OC0 operation */
        while(!(TFLG1 & C0F)); /* wait for PT0 pin to go low */
        TCTL2 = 0x03;    /* set OC0 pin action to pull high */
        TC0   += lo_time;     /* start a new OC0 operation */
```

ECE 4510/5530

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
Example 8.6: Estimating an Input Waveforms Frequency

```c
#include <hcs12.h>
#include <vectors12.h>
#include <convert.c>
#include <stdio.c>
#define INTERRUPT __attribute__((interrupt))
unsigned int frequency;
void INTERRUPT TC0_isr(void);
void main(void)
{
    char arr[7];
    char *msg = "Signal frequency is ";
    int i, oc_cnt;
    unsigned frequency;
    UserTimerCh0 = (unsigned short)&TC0_isr;
    TSCR1 = 0x90;   /* enable TCNT and fast flag clear */
    TSCR2 = 0x02;   /* set prescale factor to 4 */
    TIOS = 0x02;    /* select OC1 and IC0 */
    oc_cnt = 100;   /* prepare to perform 100 OC1 operations */
    frequency = 0;
```
Example 8.6: C Code (2 of 2)

```c
TCTL4 = 0x01; /* prepare to capture PT0 rising edge */
TFLG1 = C0F; /* clear C0F flag */
TIE |= IC0; /* enable IC0 interrupt */
asm("cli");
TC1 = TCNT + 60000;
while (oc_cnt) {
    while(!(TFLG1 & C1F));
    TC1 = TC1 + 60000;
    oc_cnt = oc_cnt - 1;
}
asm("sei");
int2alpha(frequency, arr);
puts(msg);
puts(&arr[0]);
asm("swi");
}

void INTERRUPT TC0_isr(void)
{
    TFLG1 = C0F; /* clear C0F flag */
    frequency ++;
}
```

C Program for Siren Generation (1 of 2)

```c
#include "c:\egnu091\include\hcs12.h"
#include "c:\egnu091\include\delay.c"
#define HiFreq 1250
#define LoFreq 5000

int delay;               /* delay count for OC5 operation */

int main(void)
{
    asm("cli");
    TSCR1 = 0x90;      /* enable TCNT and fast timer flag clear */
    TSCR2 = 0x03;      /* set prescaler to TCNT to 1:8 */
    TIOS |= BIT5;      /* enable OC5 */
    TCTL1 = 0x04;      /* select toggle for OC5 pin action */
    delay = HiFreq;    /* use high frequency delay count first */
    TC5   = TCNT + delay; /* start an OC5 operation */
    TIE   |= BIT5;     /* enable TC5 interrupt */
    asm("cli");
```

ECE 4510/5530

C Program for Siren Generation (2 of 2)

```c
while(1) {
    delayby100ms(5);   /* wait for half a second */
    delay = LoFreq;    /* switch to low frequency tone */
    delayby100ms(5);   /* wait for half a second */
    delay = HiFreq;    /* switch to high frequency tone */
}
return 0;
}

#pragma interrupt_handler oc5_ISR
void oc5_ISR(void)
{
    TC5 += delay;
}

#pragma abs_address:vtimch5               // Initialize the Interrupt Vector address
void (*interrupt_vectors[])(void) = {oc5_ISR};   // Assign the function pointer
#pragma end_abs_address // to the ISR
```
PWM Block Diagram

Figure 8.38 HCS12 PWM block diagram.
void main(void){
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    asm("cli");
    while(state == 1){
        printf("Hello World!\n\r");
        delayby100ms(2);
        char_in = checkchar();
        if (!(char_in ==0)){
            state = 2;
            printf("On to state 2.\n\r");
        }else{
            printf("State2\n\r");
            delayby100ms(10);
            if (count == 4){
                state = 3;
                printf("On to state 3.\n\r");
            }else{
                printf("On to state 4.\n\r");
                state = 4;
            }
        }
    }
    while(state == 4){
        asm("nop");
    }
}

init_pwm();
printf("PWM Initialized.\n\r");
printf("On to state 4.\n\r");
state = 4;
while(state == 4){
    asm("nop");
}
void init_pwm(void)
{
    PWMPRCLK = 0x44; // Prescale A and B by 16 (1.5 MHz)
    PWMCLK = 0x00; // Select A and B clocks
    PWMCTL = PSWAI | PFRZ; // All 8 bit registers, stop clocks for W or F
    PWMCAE = CAE2; // PWM.2 is center aligned
    PWMCAE &= ~(CAE4 | CAE0); // PWM.0 and PWM.4 are left aligned
    PWMPOL = PPOL0; // PWM.0 is positive polarity
    PWMPOL &= ~(PPOL2 | PPOL4); // PWM.2 and PWM.4 are negative polarity
    PWMPER0 = 150; // Left aligned 300 count period (10 kHz)
    PWMPER2 = 75; // Center aligned 2 x 150 count period (10 kHz)
    PWMPER4 = 150; // Left aligned 300 count period (10 kHz)
    PWMDTY0 = 50; // Positive 100/300 duty cycle
    PWMDTY2 = 50; // Centered Negative (150-100)/150 duty cycle
    PWMDTY4 = 100; // Negative (300-200)/300 duty cycle
    PWME = (PWME0 | PWME2 | PWME4); //Enable channel 0, 2 and 4
}
void main(void){

COPCTL=0x00;
asm("sei");
pll_init();
crg_init();
init_sci0();
DDRS |= 0x80;

asm("cli");

while(state == 1){
    printf("Hello World!\n\r");
    delayby100ms(2);
    char_in = checkchar();
    if (!(char_in ==0)){
        state = 2;
        printf("On to state 2.\n\r");
    }
}

    while(state == 2){
        printf("State2\n\r");
        delayby100ms(10);
        if (count == 4){
            state = 3;
            printf("On to state 3.\n\r");
        }
        count++;
    }

    init_pwm();
    printf("PWM Initialized.\n\r");
    printf("On to state 4.\n\r");
    state = 4;

    while(state == 4){
        asm("nop");
    }
}
void init_pwm(void) {
    PWMPRCLK = 0x33; // Prescale A and B by 8 (3 MHz)
    PWMCLK = 0x00; // Select A and B clocks
    PWMCTL |= CON45 | CON23 | CON01 | PSWAI | PFRZ; // 16 bit registers, stop clocks for W or F
    PWMCAE |= CAE3; // PWM.23 is center aligned
    PWMCAE &= ~(CAE5 | CAE1); // PWM.01 and PWM.45 are left aligned
    PWMPOL |= PPOL1; // PWM.01 is positive polarity
    PWMPOL &= ~(PPOL3 | PPOL5); // PWM.3 and PWM.5 are negative polarity
    PWMPER01 = 300; // Left aligned 300 count period (10 kHz)
    PWMPER23 = 150; // Center aligned 2 x 150 count period (10 kHz)
    PWMPER45 = 300; // Left aligned 300 count period (10 kHz)
    PWMDTY01 = 100; // Positive 100/300 duty cycle
    PWMDTY23 = 100; // Centered Negative (150-100)/150 duty cycle
    PWMDTY45 = 200; // Negative (300-200)/300 duty cycle
    PWME |= (PWME0 | PWME1 | PWME2 | PWME3 | PWME4 | PWME5); // Enable channel 01, 23 and 45
}
The HCS12 SCI Subsystem (3 of 3)

Figure 9.8 HCS12 SCI block diagram

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
```
char in_buffer[80];
char out_buffer[80];
char char_in;
char state = 1;
char count = 0;
char *rcv_ptr;
char *tx_ptr;
int in_length;

void main(void){
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    asm("cli");

    while(state == 1){
        printf("Hello World!\n\r");
        delayby100ms(2);
        char_in = checkchar();
        if (!(char_in == 0)){
            state = 2;
            printf("On to state 2.\n\r");
        }
    }

    while(state == 2){
        printf("State2\n\r");
        delayby100ms(10);
        if (count == 4){
            state = 3;
            printf("On to state 3.\n\r");
        }
        count++;
    }

    while(state == 3){
        getstr(in_buffer);
        in_length = strlen(in_buffer);
        printf(in_buffer);
        newline();
        strflip(out_buffer,in_buffer,in_length);
        printf(out_buffer);
        newline();
    }
}
```
SCI0 Functions

void init_sci0(void)
{
    // BR=ECLK/(16 x BR)
    SCI0BD = 156;
    // 24 MHz E-clock, 9600 bps desired (0.16% error)
    //SCISWAI, wakeup to mark, idle line after stop
    SCI0CR1 = SCISWAI | WAKE | ILT ;
    // 8-bit data, parity disabled, even parity
    SCI0CR1 &= ~(M | PE | PT) ;
    // transmitter enable, receiver enable,
    // no interrupts enabled, not a break bit, not wake up
    SCI0CR2 = TE | RE;
}

int getchar(void)
{
  while(!(SCI0SR1 & RDRF));
  return SCI0DRL;
}

int putchar(char cx)
{
  while(!(SCI0SR1 & TDRE))
    continue;
  SCI0DRL = cx;
  return 0;
}

char checkchar(void)
{
  if(SCI0SR1 & RDRF)
    { return SCI0DRL; }
  else
    { return 0; }
}
SCI0 Functions

int getstr(char *ptr)
{
    while (1) {
        *ptr = getchar();
        if (*ptr == 0x0D){
            *ptr = 0;
            return 0;
        } else ptr++;
    }
    return 0;
}

int newline(void)
{
    putchar(0x0D);
    putchar(0x0A);
    return 0;
}

char strflip(char *dest, const char *src, int copy_length)
{
    char *save = dest;
    int ii;

    dest = dest + copy_length;
    *dest = 0;
    dest--;

    for(ii=0; ii<copy_length;ii++){
        (*dest = *src);
        dest--;
        src++;
    }

    return 0;
}
/*******************************************************************/
/* The following function outputs a carriage return and a linefeed */
/* to move the cursor to the first column of the next row.        */
/*******************************************************************/

int newline(void)
{
    putchar(0x0D);   // Carriage Return
    putchar(0x0A);   // Line Feed
    return 0;
}
Reading Into a Buffer

/* The following function reads a string from the SCI0 port by calling the getchar function until the CR character is reached */

int getstr(char *ptr)
{
    while (1) {
        *ptr = getchar();
        if (*ptr == 0x0D){
            *ptr = 0;
            return 0;
        }
        else ptr++;
    }
    return 0;
}
# Using the stdio library

## Input-output manipulation functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Notes</th>
<th>Notes                                                                ighet</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin</td>
<td>makes a function call to getchar</td>
<td>reads and returns a character from a given stream and advances the file position indicator; it is allowed to be a macro with the same effects as fgetc, except that it may evaluate the stream more than once</td>
</tr>
<tr>
<td>stdout</td>
<td>makes a function call to putchar</td>
<td>has the same effects as getc(stdin)</td>
</tr>
<tr>
<td>gets</td>
<td>reads characters from stdin until a newline is encountered and stores them in its only argument</td>
<td>reads and returns a character to a stream and advances the file position indicator for it; equivalent to fputc, except that a macro version may evaluate the stream more than once</td>
</tr>
<tr>
<td>puts</td>
<td>outputs a character string to stdout</td>
<td>has the same effects as putc(stdout)</td>
</tr>
</tbody>
</table>

**print**

used to print to the standard output stream

**sprintf**

used to print to a char array (C string)

**putc**

writes and returns a character to a stream and advances the file position indicator for it; equivalent to fputc, except that a macro version may evaluate the stream more than once

**putchar**

has the same effects as putc(stdout)

**puts**

outputs a character string to stdout

**scanf,**

used to input from the standard input stream

**sscanf,**

used to input from a char array (e.g., a C string)
SS Multiple IC Interconnection

Figure 10.9 Single-master and multiple-slave device connection (method 1)
void main(void){

COPCTL=0x00;
asm("sei");
pll_init();
crg_init();
init_spi0();
// Designed to test loop back of SPI0
while(state == 2){
    tx0_ptr = &data_buffer1[0];
    rcv0_ptr = &in_buffer1[0];

    while (!(*tx0_ptr==0)){
        spi0_select;  // SPI Exchange
        *rcv0_ptr=exchspi0(*tx0_ptr);
        tx0_ptr++;
        rcv0_ptr++;
    }
    *rcv0_ptr = 0;
    printf(in_buffer1);
    newline();
    rcv0_ptr = &in_buffer1[0];
    for(ii=0;ii<80;ii++){
        *rcv0_ptr=0;
        rcv0_ptr++;
    }
    delayby1us(5);
} // End of main()
void init_spi0(void)
{
    // SPI0 port pin name definitions (not needed, but nice)
    #define MISO0 PTS4;
    #define MOSI0 PTS5;
    #define SCK0 PTS6;
    #define SS0n PTS7;          // Make a parallel pin. Explicit sw operation.
    // SPI0 port interface macros
    #define spi0_select PTS &= ~PTS7;
    #define spi0_deselect PTS |= PTS7;

    SPI0BR = 0x20;          //set baud rate to 24 MHz/6 = 4 MHz
    SPI0CR1 |= SPE | MSTR | CPHA;         //Enable, Master, SCLK high active low, even phases
    SPI0CR1 &= ~(S SOE | CPOL );      //do not use the SSn output pin
    SPI0CR2 |= SPISWAI;              //stop selk in wait mode
    SPI0CR2 &= ~(MODFEN | SPC0);     //do not use the SSn output pin, normal MISO MOSI pins

    DDRS |= PTS7;       //SS0n parallel output pin

    PERS |= PERS4;       //enable pullup/down on SCI0 MISO pin
    PERS &= ~(PERS5 | PERS6 | PERS7);  //disable pullup/down on SCI0 pins
    PPSS &= ~(PPSS4);    //pull-ups on SCI0 MISO
    WOMS |= WOMS4;      //open-drain drive on SCI0 MISO
}

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
char exchcspi0(char cx)
{
    char temp;
    while(!(SPI0SR & SPTEF)); /* wait until write is permissible */
    SPI0DR = cx; /* output the byte to the SPI */
    while(!(SPI0SR & SPIF)); /* wait until write operation is complete */
    return SPI0DR; /* return the character and clear SPIF flag */
}
putcspi0 and getcspi0

/******************************************************************************/
/* The following function outputs a character to the SPI0 interface.         */
/******************************************************************************/
void putcspi0 (char cx)
{
    char temp;
    while(!(SPI0SR & SPTEF));  /* wait until write is permissible */
    SPI0DR = cx;               /* output the byte to the SPI */
    while(!(SPI0SR & SPIF));   /* wait until write operation is complete */
    temp = SPI0DR;            /* clear SPIF flag */
}

/******************************************************************************/
/* The following function reads a character from the SPI0 interface.         */
/******************************************************************************/
char getcspi0(void)
{
    while(!(SPI0SR & SPTEF));  /* wait until write is permissible */
    SPI0DR = 0x00;             /* trigger 8 SCK pulses to shift in data */
    while(!(SPI0SR & SPIF));   /* wait until a byte has been shifted in */
    return SPI0DR;            /* return the character and clear SPIF flag */
}
void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    while(state == 1) {
        printf("Hello World!\n\r");
        delayby100ms(10);
        char_in = checkchar();
        if (!char_in == 0) {
            state = 2;
            printf("On to state 2.\n\r");
            state = 3;
            printf("On to state 3.\n\r");
        }
    }
    printf(data_buffer1);
    printf(data_buffer2);
}

void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    void main(void) {
    COPCTL=0x00;
    asm("sei");
    pll_init();
    crg_init();
    init_sci0();
    init_spi0();
    init_spi1();
    asm("cli");
    // Designed to test SPI0-Master to SPI1-Slave
    // Note: this only works for 8-bit transfers
    while(state == 3) {
        state = 3;
        printf("On to state 3.\n\r");
        if (!(*tx0_ptr==0)) {
            spi0_deselect;
            *rcv0_ptr=0;
        }
    }
    // Transfer the data
    while (!(*tx0_ptr==0)) {
        putcspi1(*tx1_ptr);
        spi0_deselect;
        *rcv0_ptr=0;
    }
    // Print the master and slave input buffers
    printf("\nMaster Input Buffer--\n");
    printf(in_buffer1);
    printf("\nSlave Input Buffer--\n");
    printf(in_buffer2);
    printf("\n\r");
    printf(data_buffer1);
    printf(data_buffer2);
}

// Designed to test SPI0-Master to SPI1-Slave
// Note: this only works for 8-bit transfers
while(state == 3) {
    state = 3;
    printf("On to state 3.\n\r");
    if (!(*tx0_ptr==0)) {
        spi0_deselect;
        *rcv0_ptr=0;
    }
}
void init_spi1(void)
{
    // spi1 port pin name definitions (not needed, but nice)
    #define MISO1 PH0;
    #define MOSI1 PTH1;
    #define SCK1 PTH2;
    #define SS1n PTH3;
    // spi1 port initialization
    SPI1BR = 0x20; //set baud rate to 24 MHz/6 = 4 MHz
    SPI1CR1 |= SPE | CPHA; //Enable, Master, SCLK high active low, even phases
    SPI1CR1 &= ~(SOE | MSTR | CPOL ); //do not use the SSn output pin
    SPI1CR2 |= SPISWAI; //stop sclk in wait mode
    SPI1CR2 &= ~(MODFEN | SPC0); //do not use the SSn output pin, normal MISO MOSI pins
}
**putcspi1** and **getcspi1**

```c
void putcspi1 (char cx) {
    char temp;
    while(!(SPI1SR & SPTEF)); /* wait until write is permissible */
    SPI1DR = cx;              /* output the byte to the SPI */
}

char getcspi1(void) {
    while(!(SPI1SR & SPIF)); /* wait until a byte has been shifted in */
    return SPI1DR; /* return the character and clear SPIF flag */
}
```

---

ECE 4510

DAC AD7390

FEATURES
- Micropower—100 uA
- Single-Supply—2.7 V to 5.5 V Operation
- Compact 1.75 mm Height SO-8 Package and 1.1 mm Height TSSOP-8 Package
- AD7390—12-Bit Resolution
- SPI and QSPI Serial Interface Compatible with Schmitt Trigger Inputs

APPLICATIONS
- Automotive 0.5 V to 4.5 V Output Span Voltage
- Portable Communications
- Digitally Controlled Calibration
Example DAC AD7390

```c
void dac_output(unsigned int voltage) // output voltage
{
    char ms_byte;
    char ls_byte;

    ms_byte = (voltage>>8) & 0x0F; // Shift 16-bit value to 8 lsbs, and upper 4 bits
    ls_byte = (voltage) & 0xFF; // Convert 16-bit to 8 bit

dac_select;
spi0_transmit( ms_byte ); // D11-D8: transmit the most sig. byte
spi0_transmit( ls_byte ); // D7-DB0: transmit the least sig. byte
dac_deselect;
}

void dac_clear(void) // zero output voltage
{

dac_select; // LDn goes high
dac_clear_select; // CLRn goes low
asm("nop"); // clock cycle delay for clearing
dac_clear_deselect // CLRn returns high
dac_deselect; // LDn returns low
}
```

ECE 4510/5530 Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
AD7705 ADC

- **FEATURES**
- AD7705: 2 fully differential input channel ADCs
  - 16 bits no missing codes, 0.003% nonlinearity
  - Programmable gain front end: gains from 1 to 128
- 3-wire serial interface
  - SPI®, QSPI®, MICROWIRE®, and DSP-compatible
  - Schmitt-trigger input on SCLK
- Ability to buffer the analog input
- 2.7 V to 3.3 V or 4.75 V to 5.25 V operation
- Power dissipation 1 mW maximum @ 3 V
- Standby current 8 μA maximum
- 16-lead PDIP, 16-lead SOIC, and 16-lead TSSOP packages

![FUNCTIONAL BLOCK DIAGRAM](image)
AD7705.c elements

```c
#define adc_select      PTX &= ~ADC_CSn;
#define adc_deselect    PTX |= ADC_CSn;

void adc_reset()    //reset the ADC
{
    // Software Reset
    adc_select;
    adcspi_transmit(0xFF);
    adcspi_transmit(0xFF);
    adcspi_transmit(0xFF);
    adcspi_transmit(0xFF);
    adc_deselect;
}
```
AD7705.c elements

```c
void adc_init()    //reset the ADC
{
    // Initialize clock
    adc_select;
    adcspi_transmit(CLOCK_REG); // Select clock register for write, device enabled, register pair 0
    adcspi_transmit(FS25);       // CLKDIV=0, CLK is 1 MHz (0), 25 Hz (40 msec)
    adc_deselect;
    // Initialize setup
    adc_select;
    adcspi_transmit(SETUP_REG); // Select setup register
    adcspi_transmit(0x00);       // Mode=Normal, Gain = 1 (000), Bipolar (0), FSYNC=0
    adc_deselect;
    while(PORTC & 0x10);        /* wait for /DRDY to go low */
    // Run Self-Cal
    adc_select;
    adcspi_transmit(SETUP_REG); // Select setup register
    adcspi_transmit(0x40);       // Mode=Self Cal
    adc_deselect;
    while(PORTC & 0x10);        /* wait for /DRDY to go low */
    // Place in Standby
    adc_select;
    adcspi_transmit(STANDBY);    // Select comm register for write, standby, register pair 0
    adc_deselect;
}
```

int adc_convert 1() //reset the ADC
{
    unsigned int temp0, temp1, ADC_value;
    // Start ADC
    adc_select;
    adcspi_transmit(COMM_REG); // Select comm register for write, device enabled, register pair 0
    adc_deselect;
    while(PORTC & 0x10); /* wait for /DRDY to go low */

    // Read Data
    adc_select;
    adcspi_transmit( DATA_REG | READ_REG); // Select data reg. for read
    temp1 = adcspi_exchange(0x00);
    temp0 = adcspi_exchange(0x00);
    adc_deselect;

    ADC_value = (temp1<<8) | temp0;
    return(ADC_value);

    // Place in Standby
    adc_select;
    adcspi_transmit(STANDBY); // Select comm register for write, standby, register pair 0
    adc_deselect;
}

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
AD7705.h elements

- Compose a register read/write request by oring the fields together

```c
void adc_setup_norm(void)    //setup the ADC
{
    adc_select;
    adcspi_transmit(SELF_SETUP_REG|WRITE_REG|CH0);
    adcspi_transmit(GAIN0);
    adc_deselect;
}
```
## AD7705.h elements

```c
#define adc_select                  PTX  &!= ~ADC_CSn;
#define adc_deselect                PTX  |= ADC_CSn;

#define MD_NORM                     0x00;
#define MD_SCAL                      0x40;
#define MD_ZCAL                      0x80;
#define MD_FCAL                      0xC0;

#define GAIN1                       0x00;
#define GAIN2                       0x08;
#define GAIN4                       0x10;
#define GAIN8                       0x18;
#define GAIN16                      0x20;
#define GAIN32                      0x28;
#define GAIN64                      0x30;
#define GAIN128                     0x30;

#define BIPOLAR                     0x00;
#define UNIPOLAR                    0x04;

#define BUFFON                      0x02;
#define BUFFOFF                     0x00;
#define FSYNC                       0x01;
```

- Compose a register read/write request by oring the fields together

```c
void adc_setup_selfcal(void)    //self cal the ADC
{
    adc_select;
    adcspi_transmit(SETUP_REG|WRITE_REG|CH0)
    adcspi_transmit(MD_SCAL |GAIN0|BUFFON);
    adc_deselect;
}
```

- You may want to wait for ready and then repeat for CH1
AD7705.h elements

#define adc_select    PTX &= ~ADC_CS;
#define adc_deselect   PTX |= ADC_CS;
#define CLKDISABLE    0x10;
#define CLKDIV        0x08;
#define CLK2.4MHZ      0x04;
#define FS20          0x00;
#define FS25          0x01;
#define FS100         0x02;
#define FS200         0x03;
#define FS50          0x00;
#define FS60          0x01;
#define FS250         0x02;
#define FS500         0x03;

- Compose a register read/write request by oring the fields together

void adc_set_clock (void)  //clock definition for ADC
{
    adc_select;
    adcspi_transmit(CLOCK_REG|WRITE_REG);
    adcspi_transmit(FS25);
    adc_deselect;
}
Inter-Integrated Circuit (I2C)

- Inter-Integrated Circuit (I2C) Interface
  - Synchronous (fast, but with pull-ups required)
  - More than 8-bits at a time
  - Multiple devices can communicate using the same serial lines
  - Bi-directional data transfer
  - Does not require a single master
Byte Level Transfer

- Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must be followed by an Acknowledge bit. Data is transferred with the Most Significant Bit (MSB) first (see Figure 6). If a slave cannot receive or transmit another complete byte of data until it has performed some other function, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.

NXP, I2C-bus specification and user manual, Rev. 5 — 9 October 2012

ECE 4510/5530
85

Material from or based on: The HC12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
IIC Block Diagram

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
I2C Initialization

- Compute an appropriate value and write it into the IBFD register.
- Load a value into the IBAD register if the MCU may operate in slave mode.
- Set the IBEN bit of the IBCR register to enable I2C module.
- Modify the bits of the IBCR register to select master/slave mode, transmit/receive mode, and interrupt enable mode

```c
void openI2C (char ibc, char i2c_ID)
{
    IBCR = IBEN; /* enable I2C module, slave receive */
    IBFD = 0x1F; /* set up I2C baud rate */
    IBAD = i2c_ID; /* set up slave address */
    IBCR &= ~IBIE; /* disable I2C interrupt */
    IBCR |= IBSWAI; /* disable I2C in wait mode */
}
```
I2C Master Addressing

- Check to insure the bus is not busy (IBSR bit IBB).
- Send start condition as master (IBCR bits MSSL and Transmit)
- Send address and wait for the transmission to complete.
- Modify the bits of the IBCR register to select master/slave mode, transmit/receive mode, and interrupt enable mode

```c
char sendSlaveID (char cx char RdWrn)
{
    while (IBSR&IBB); /* wait until I2C bus is idle */
    IBCR |= TXRX+MSSL; /* generate a start condition */
    IBDR = cx | RdWrn; /* send out the slave address with R/W bit */
    while(!(IBSR & IBIF)); /* wait for address transmission to complete */
    IBSR = IBIF; /* clear IBIF flag */
    return(IBSR & RXAK); /* return the status of the acknowledge */
}
```
I2C Master Sending Byte

- Check to insure the address acknowledge was received
- Send data and wait for the transmission to complete.
- Send Stop Condition

```c
if(sendSlaveID(address_isc_device 0x00)) {
    IBCR &= ~MSSL; /* generate stop condition */
    error_condition ....
} else {
    IBDR  = write_data; /* send out the value cx */
    while (!(IBSR & IBIF)); /* wait until the byte is shifted out */
    IBSR   = IBIF; /* clear the IBIF flag */
    if(IBSR & RXAK) ACK_ERR = 1; /* status of the acknowledge */
    IBCR &= ~MSSL; /* generate stop condition */
}
```
I2C Master Reading Byte

- Check to insure the address acknowledge was received
- Place device in receive mode and perform “dummy read”
- Receive and provide NACK to end communications.
- Send Stop Condition

```c
if(sendSlaveID(address_isc_device 0x01))
{
    IBCR &= ~MSSL; /* generate stop condition */
    error_condition ....
}
else
{
    IBCR &= ~(TXRX + TXAK); /* prepare to receive and acknowledge */
    IBCR |= TXAK; /* prepare to not acknowledge */
    dummy = IBDR; /* a dummy read */
    while(!(IBSR & IBIF)); /* wait for the byte to shift in */
    IBSR = IBIF; /* clear the IBIF flag */
    IBCR &= ~MSSL; /* generate stop condition */
    buf = IBDR; /* place the received byte in buf */
}
```

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
I2C Real-Time Clock/Calendar
MCP7940M

Features

• Real-Time Clock/Calendar (RTCC):
  – Hours, Minutes, Seconds, Day of Week, Day, Month and Year
  – Dual alarm with single output

• On-Chip Digital Trimming/Calibration:
  – Range -127 to +127 ppm, Resolution 1 ppm

• Programmable Open-Drain Output Control:
  – CLKOUT with 4 selectable frequencies
  – Alarm output

• 64 Bytes SRAM

• Low-Power CMOS Technology:
  – Dynamic Current: 400 μA max write

• 100 kHz and 400 kHz Compatibility

• ESD Protection >4,000V

• Packages include 8-Lead SOIC, TSSOP, 2x3 TDFN, MSOP and PDIP

• Temperature Ranges:
  – Industrial (I): -40°C to +85°C

Typical Application Schematic

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
Lalith’s Example Code

- **I2C routines**
  - i2c.h and i2c.c
  - // Function prototypes
    - int I2Cinit (char baud_divisor, char dev_id);
    - int I2Cstart (char);
    - int I2Crestart (char);
    - int I2Cwritec (char data);
    - int I2Cwritestr (char *,int);
    - char I2Creadc_ack (void);
    - char I2Creadc_nack (void);
    - int I2Creadstr (char *,int);
  - // uses “special definitions” for
    - I2C_STOP IBCR
    - I2C_TX_ACK IBCR
    - I2C_TX_NACK IBCR
    - I2C_ACK_CHK(x)

- **Real Time Clock/Calendar**
  - rtic.h and rtic.c
  - // Function prototypes
    - int RTICinit (void);
    - int RTICwrite (char,char);
    - char RTICread (char);
    - int RTICpwm (void);
  - //global declaration externs

Material from or based on: The HCS12/9S12: An Introduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.
CAN Physical Layer

- Data Frames are transmitted on a two-wire common bus as CAN high (CAN_H) and CAN low (CAN_L) signals.

Figure 1 — Suggested electrical interconnection
CAN may sound scary, but …

• The CAN controller will perform the majority of the tasks described.
  – It must be initialized.
  – It must be told which IDs to monitor and send.
  – It must watch for RTRs

• The CAN transceiver will take care of outputting and monitoring the correct voltage levels.

• Our software will have to monitor status bits (respond to interrupts), send and receive data blocks (0-8 Bytes).
main()
{
    unsigned char ii;
    COPCTRL=0x00;
    asm ("sei"); //global interrupt disable
    pll_init();
    crg_init(); //Initialize SCI0
    init_sci0(); //Initialize SCI0
    can0_init(); //Initialize CAN0
    can1_init(); //Initialize CAN0
    printf("CAN0 Initialized.\n\r");
    printf("CAN1 Initialized.\n\r");
    for (ii=0;ii<64;ii++) {
        rec0[ii]=0;
        rec1[ii]=0;
    }
    asm("cli"); //global interrupt enable
    state = 5;
}

while(1) {
    if(state==1) {
        state=0;
        tx_status = 0;
        printf("Message 1 Sent.\n\r");
    }
    if(state==2) {
        state=0;
        printf("Message 2 received.\n\r");
        printf(&rec0[0]);
        newline();
        for (ii=0;ii<64;ii++) {
            rec0[ii]=0;
        }
    }
    if(state==3) {
        state=0;
        tx_status = 0;
        printf("Message 2 Sent.\n\r");
    }
    if(state==4) {
        state=0;
        printf("Message 1 received.\n\r");
        printf(&rec1[0]);
        newline();
        for (ii=0;ii<64;ii++) {
            rec1[ii]=0;
        }
    }
    if(state==5) {
        state = 0;
        tx_status = 1;
        printf("Start Sending.\n\r");
        can0_tx_start();
    }
    if(state==6) {
        state = 0;
        tx_status = 2;
        printf("Start Sending.\n\r");
        can1_tx_start();
    }
    if((state==0)&(tx_status==0)) {
        delayby1ms(500);
        if(last_message==0) {
            last_message=1;
            state = 6;
        } else {
            last_message=0;
            state = 5;
        }
    }
    if(last_message==0) {
        last_message=1;
        state = 6;
    } else {
        last_message=0;
        state = 5;
    }
}
return(0); // end of main()
void can0_init(void)
{
    //enable CAN and CANE = 1, MSCAN clock source is the Oscillator clock (16MHz)
    CAN0CTL1 = CANE;
    CAN0CTL0 = INITRQ; //initialize the CAN

    while((CAN0CTL1 & INITAK)==0x00)
    {
        asm("nop");
    } // Waiting for the acknowledge the init.

    CAN0BTR0 = 0x01; //Baud Rate Prescaler for 8 MHz TQ clock (16 MHz/2).
    CAN0BTR0 &= ~(SJW1 | SJW0); //synchronization jump width are one (SJW = 1Tq).
    CAN0BTR1 |= (TSEG21 | TSEG11 | TSEG10);
    //one sample per bit, TSEG2 = 3 Tq, TSEG1+PROPSEG = 4 Tq
    //8Tq = 1 Sync_seg + 3 TSEG2 + 4 (TSEG1+PROPSEG)
    //supports 1 Mbps using a 16 MHz crystal and 8 MHz OSCCLOCK
    CAN0IDAC |= IDAM0;                  //four 16-bit acceptance filters
    CAN0IDMR0 = 0x00; //11-bit address masks
    CAN0IDMR1 = 0x1F;                     CAN0IDMR2 = 0x00;                     CAN0IDMR3 = 0x1F;                     CAN0IDMR4 = 0x00;                     CAN0IDMR5 = 0x1F;                     CAN0IDMR6 = 0x00;
    CAN0IDMR7 = 0x1F;
    CAN0BTR1 = 0x23; //Class default

    while((CAN0CTL0 & SYNCH) == 0x00)
    {
        asm("nop");
    } //waiting for the synchronization

    CAN0CTRL0 &= ~(INITRQ); //turn-off the initialization state
    CAN0CTRL0 &= ~ (INITRQ);

    //enable CAN and CANE = 1, MSCAN clock source is the Oscillator clock (16MHz)
    CAN0TIER = TXEIE0; //enable the transmit interrupt buffer 0
    CAN0RIER = RXFIE; //enable the receive interrupt
    CAN0TBSEL = TX0; //select the transmit buffer 0
}
void can0_tx_start(void)
{
    //Variables
    extern unsigned char trans0[];
    extern unsigned char trans0buff[];
    extern unsigned char trans0_idx;
    extern unsigned char state;

    unsigned char trans0_loop;
    unsigned char trans0_pos;

    while(!(CAN0TFLG&TXE0)) // wait until buffer empty
    {asm("nop");}

    trans0buff[0] = 0x5F;
    trans0buff[1] = 0xE7;
    trans0_loop=0x00;
    trans0_pos=0x08;

    while(trans0_loop<0x08)  //Send 8 characters at a time
    {
        trans0buff[trans0_loop+4] = trans0[trans0_idx];  //Fill the transmit buffer
        if (trans0[trans0_idx]==0x00) //Check if final 8 char set is partially full
            {trans0_idx=0;trans0_pos=trans0_loop+1;trans0_loop=0x08;state=0;}
        else
            {trans0_idx++;}
        trans0_loop++;
    }

    trans0buff[12] = trans0_pos;  //set the transmission length
    CAN0TFLG = TXE0;  //clear the TXE0 flag
    CAN0TIER = TXEIE0;  //enable the transmit interrupt buffer 0
}
#pragma interrupt_handler can0_tx_isr

void can0_tx_isr(void)
{

    // Variables
    extern unsigned char trans0[];
    extern unsigned char trans0buff[];
    extern unsigned char trans0_idx;
    extern unsigned char state;

    unsigned char trans0_loop;
    unsigned char trans0_pos;

    trans0buff[0] = 0x5F;
    trans0buff[1] = 0xE7;

    trans0_loop = 0x00;
    trans0_pos = 0x08;
    while(trans0_loop < 0x08) // Send 8 characters at a time
    {
        trans0buff[trans0_loop + 4] = trans0[trans0_idx]; // Fill the transmit buffer
        if (trans0[trans0_idx] == 0x00) // Check if final 8 char set is partially full
        {
            trans0_idx = 0;
            trans0_pos = trans0_loop + 1;
            trans0_loop = 0x08;
            state = 1;
            CAN0TIER &= ~TXEIE0;
        }
        else
        {
            trans0_idx++;
        }
        trans0_loop++;
    }

    trans0buff[12] = trans0_pos; // Set the transmission length
    CAN0TFLG = TXE0; /* Clear the TXE0 flag */
}
#pragma interrupt_handler can0_rcv_isr
void can0_rcv_isr(void)
{
    //Variables
    extern unsigned char rec0[];
    extern unsigned char rec0buff[];
    extern unsigned char rec0_idx;
    extern unsigned char state;
    extern unsigned char tx_status;

    unsigned char rec0_loop;

    for (rec0_loop=0;rec0_loop<rec0buff[12];rec0_loop++)
    {
        rec0[rec0_idx]=rec0buff[rec0_loop+4];
        if (rec0[rec0_idx]==0x00){
            rec0_idx=0; state=2;
        } else {
            rec0_idx++;
        }
    }
    CAN0RFLG = 0x01;                      //clear the flag
}
Procedure for Message Transmission

- **Step 1**
  - Identifying an available transmit buffer by checking the TXEx flag associated with the transmit buffer. (CANxTFLG)

- **Step 2**
  - Setting a pointer to the empty transmit buffer by writing the CANxTFLG register to the CANxTBSEL register. This makes the transmit buffer accessible to the user. (Moves it to the foreground.)

- **Step 3**
  - Storing the identifier, the control bits, and the data contents into one of the foreground transmit buffers.

- **Step 4**
  - Flagging the buffer as ready by clearing the associated TXE flag.
void can_transmit( void )
{
    static unsigned int buf_addr[3] = {0xFFFF, 0xFFFF, 0xFFFF};
    // Check if the incoming address has already been configured in a mailbox
    if( can.address == buf_addr[0] ){
        // Mailbox 0 setup matches our new message
        // Write to TX Buffer 0, start at data registers, and initiate transmission
        while(!(CAN0TFLG & TXE0)){asm("nop");}
        CAN0TBSEL = TX0;
    }
    else if( can.address == buf_addr[1] ){
        // Mailbox 1 setup matches our new message
        // Write to TX Buffer 1, start at data registers, and initiate transmission
        while(!(CAN0TFLG & TXE1)){asm("nop");}
        CAN0TBSEL = TX1;
    }
    else if( can.address == buf_addr[2] ){
        // Mailbox 2 setup matches our new message
        // Write to TX Buffer 2, start at data registers, and initiate transmission
        while(!(CAN0TFLG & TXE2)){asm("nop");}
        CAN0TBSEL = TX2;
    }
    else
    
    
}
Using multiple transmit buffers (2)

else{
    // Check if we've got any un-setup mailboxes free and use them
    // Otherwise, find a non-busy mailbox and set it up with our new address
    if( buf_addr[0] == 0xFFFF ){
        // Mailbox 0 is free
        // Write to TX Buffer 0, start at address registers, and initiate transmission
        CAN0TBSEL = TX0;
        buf_addr[0] = can.address;
    }

    else if( buf_addr[1] == 0xFFFF ){
        // Mailbox 1 is free
        // Write to TX Buffer 1, start at address registers, and initiate transmission
        CAN0TBSEL = TX1;
        buf_addr[1] = can.address;
    }

    else if( buf_addr[2] == 0xFFFF ){
        // Mailbox 2 is free
        // Write to TX Buffer 2, start at address registers, and initiate transmission
        while(!(CAN0TFLG & TXE2)){asm("nop");}
        CAN0TBSEL = TX2;
        buf_addr[2] = can.address;
    }

    else
Using multiple transmit buffers (3)

```c
else
{
    // No mailboxes free, wait until at least one is not busy
    while(( CAN0TFLG & 0x07 ) == 0x00) asm("nop");
    // Is it mailbox 0?
    if(( CAN0TFLG & TXE0 ) == TXE0) {
        // Setup mailbox 0 and send the message
        CAN0TBSEL = TX0;
        buf_addr[0] = can.address;
    }
    // Is it mailbox 1?
    else if(( CAN0TFLG & TXE1 ) == TXE1) {
        // Setup mailbox 1 and send the message
        CAN0TBSEL = TX1;
        buf_addr[1] = can.address;
    }
    // Is it mailbox 2?
    else if(( CAN0TFLG & TXE2 ) == TXE2) {
        // Setup mailbox 2 and send the message
        CAN0TBSEL = TX2;
        buf_addr[2] = can.address;
    }
}
```

Wait until a mailbox is not busy
Using multiple transmit buffers (4)

// No matches in existing mailboxes
// No mailboxes already configured, so we'll need to load an address - set it up
CAN0TXIDR0 = (unsigned char)(can.address >> 3);
CAN0TXIDR1 = (unsigned char)(can.address << 5);
CAN0TXIDR3 = 0x00; // EID8
CAN0TXIDR4 = 0x00; // EID0
CAN0TXDLR  = 0x08; // DLC = 8 bytes

} // Fill data into buffer, it's used by any address
// Allow room at the start of the buffer for the address info if needed
CAN0TXDSR0 = can.data.data_u8[0];
CAN0TXDSR1 = can.data.data_u8[1];
CAN0TXDSR2 = can.data.data_u8[2];
CAN0TXDSR3 = can.data.data_u8[3];
CAN0TXDSR4 = can.data.data_u8[4];
CAN0TXDSR5 = can.data.data_u8[5];
CAN0TXDSR6 = can.data.data_u8[6];
CAN0TXDSR7 = can.data.data_u8[7];

CAN0TFLG |= CAN0TBSEL; // clear TXE flag to send

Load the data and send
Receive Procedure

• When a valid message is received at the background receive buffer, it will be transferred to the foreground receive buffer and the RXF flag will be set to 1. (CANxRFLG & RXF)

• The user’s program has to read the received message from the RxFG and then clear the RXF flag to acknowledge the interrupt and to release the foreground receive buffer.

• When all receive buffers in the FIFO are filled with received messages, an overrun condition may occur. (CANxRFLG & OVRIF)
Using receive buffers (1)

```c
void can_receive( void )
{
    unsigned char flags;
    // Read out the interrupt flags register
    flags = CAN0RFLG;
    // Check for errors
    if(( flags & CSCIF ) != 0x00 ){
        // Clear error flags
        CAN0RFLG &= ~(CSCIF);
    }
```

Check if CSCIF error conditions to high
Using receive buffers (2)

// No error, check for received messages
else if(( flags & RXF ) != 0x00 )
{
    // Read in the info, address & message data
    can.address = CAN0IDAR0; can.address = can.address << 3;
    temp = CAN0IDAR1 >> 5; can.address = can.address | temp;
    // Fill out return structure
    // check for Remote Frame requests and indicate the status correctly
    if(( CAN0IDAR1 & RTR ) == 0x00 ){
        // We've received a standard data packet
        can.status = CAN_OK;
        // Fill in the data
        can.data.data_u8[0] = CAN0RXDSR0; can.data.data_u8[1] = CAN0RXDSR1;
        can.data.data_u8[2] = CAN0RXDSR2; can.data.data_u8[3] = CAN0RXDSR3;
        can.data.data_u8[4] = CAN0RXDSR4; can.data.data_u8[5] = CAN0RXDSR5;
        can.data.data_u8[6] = CAN0RXDSR6; can.data.data_u8[7] = CAN0RXDSR7;
        can.length = CAN0RXDLR;
    }
    else{ // We've received a remote frame request
        // Data is irrelevant with an RTR
        can.status = CAN_RTR;
    }
    // Clear the IRQ flag
    CAN0RFLG &= ~(RXF);
}
Using receive buffers (3)

```c
else{
    can.status = CAN_ERROR;
    can.address = 0x0001;
    can.data.data_u8[0] = flags; // CAN0RFLG values returned
}
```

Not CSCIF or RXF. Another error, possibly an overrun error (all receive buffers full).