Department of Electrical and Computer Engineering
College of Engineering and Applied Sciences
Western Michigan University
1903 · 2003

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.

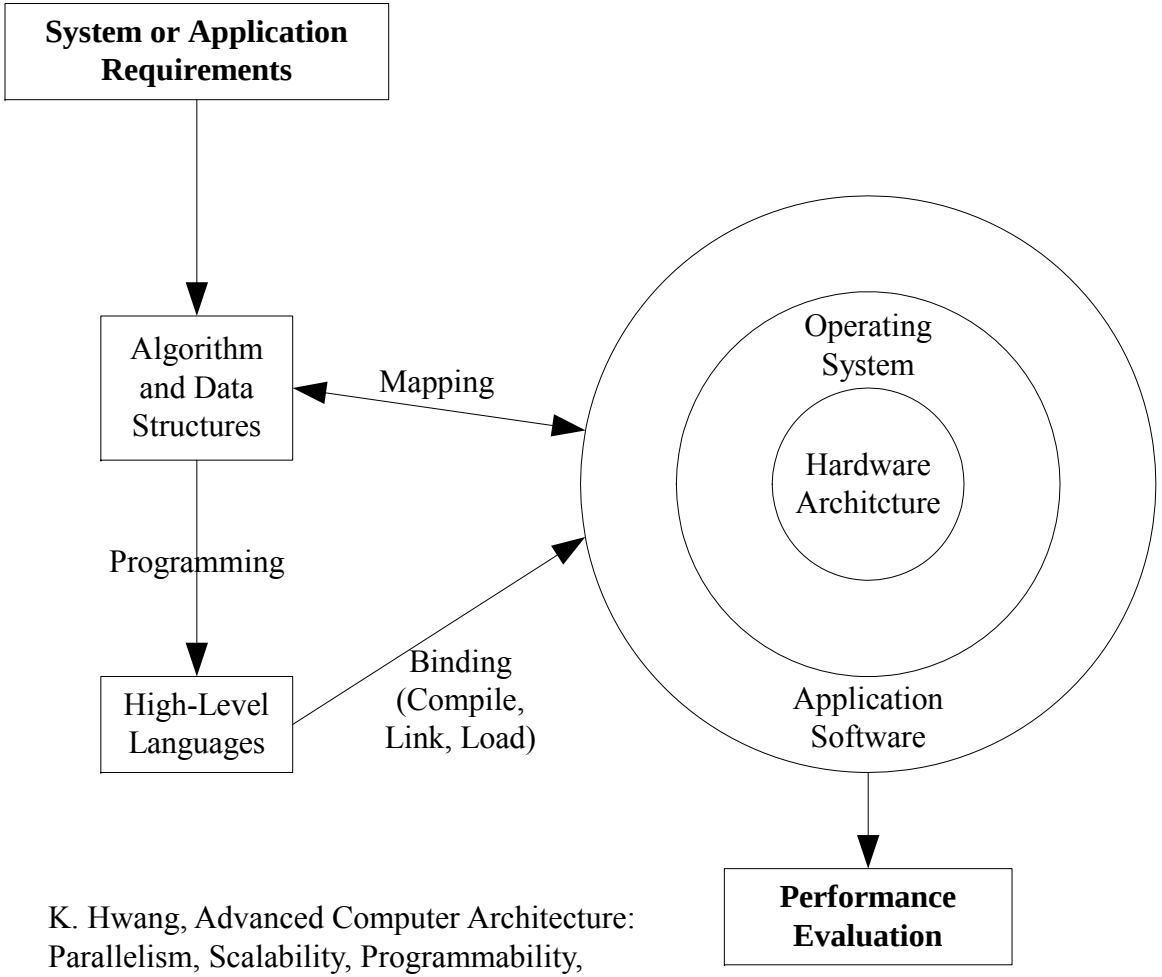
Modern Computers

- 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.

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

The Computing Problem



K. Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw Hill, 1993. ISBM: 0-07-031622-8

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

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

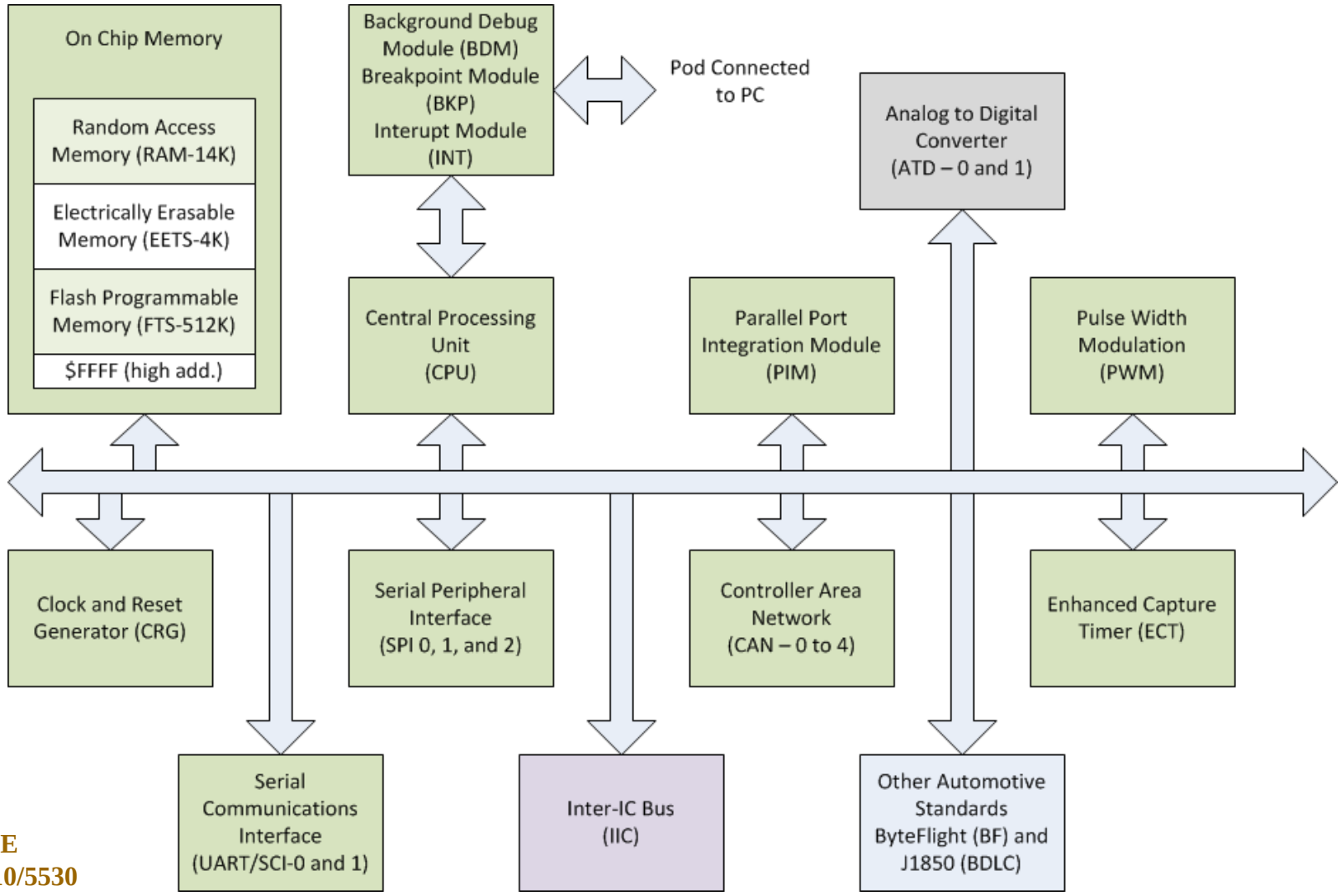
Software Development Issues

- 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

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

The CPU and Peripherals: Tools Available for Software



Interfacing, Thomson Delmar Learning, 2006.

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

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

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"

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

Code Flow Loop Diagrams

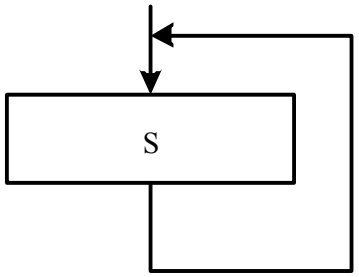


Figure 2.4 An infinite loop

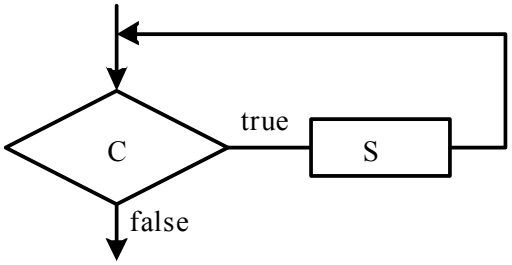
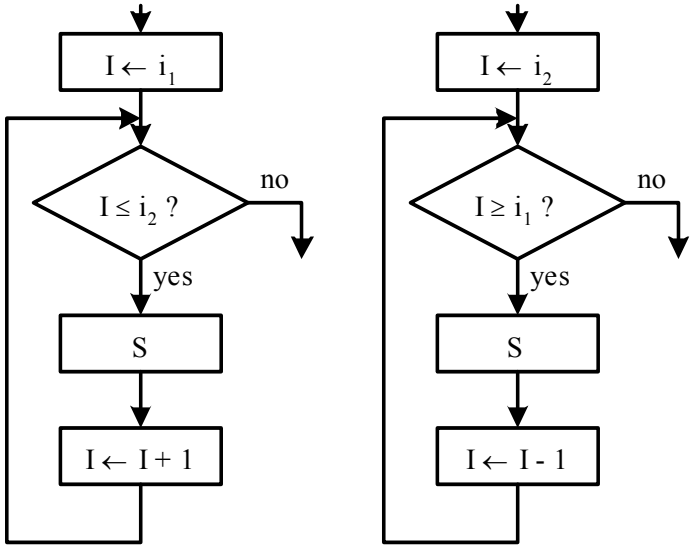


Figure 2.6 The While ... Do looping construct



(a) For I = i_1 to i_2 DO S (b) For I = i_2 downto i_1 DO S

Figure 2.5 For looping construct

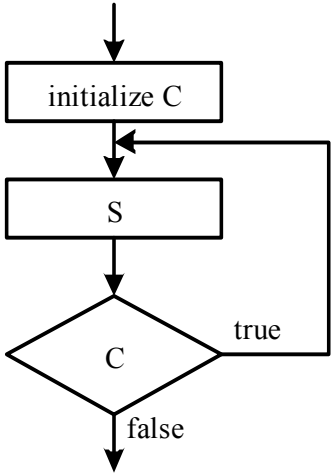
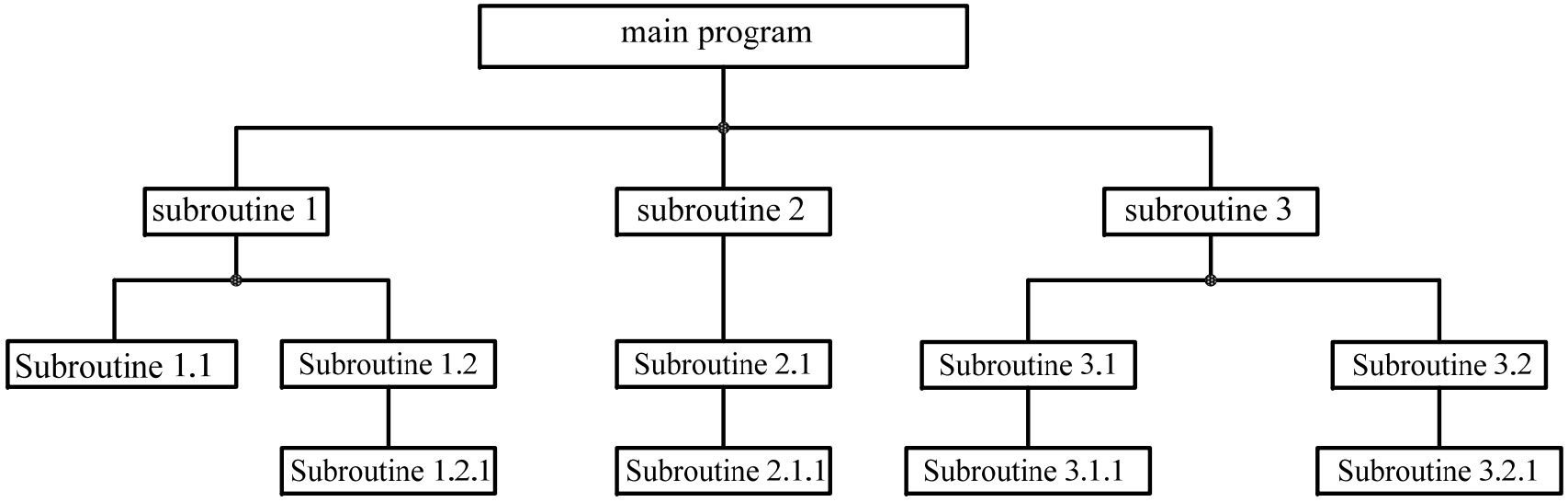


Figure 2.7 The Repeat ... Until looping construct

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



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

Interrupts For Embedded Processing

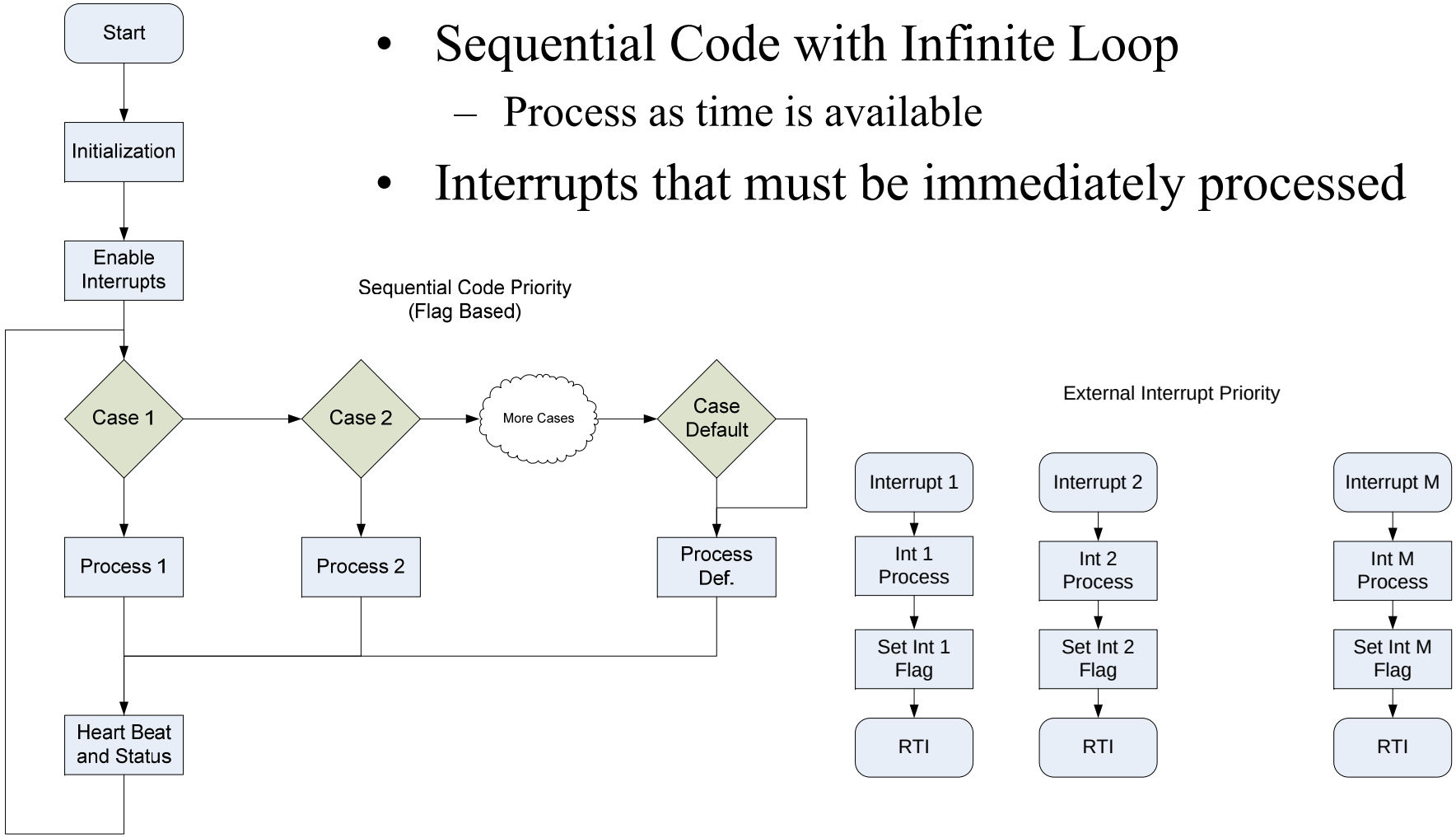
- Initialize the interrupting function and set the local interrupt enable bit
  - clear pending interrupts if possible
- Make sure a valid interrupt service routine (ISR) exists
- Make sure the system can find the ISR code
  - A ISR address table is often used for defined types of interrupts
  - Also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

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

General Real-Time Code Flow

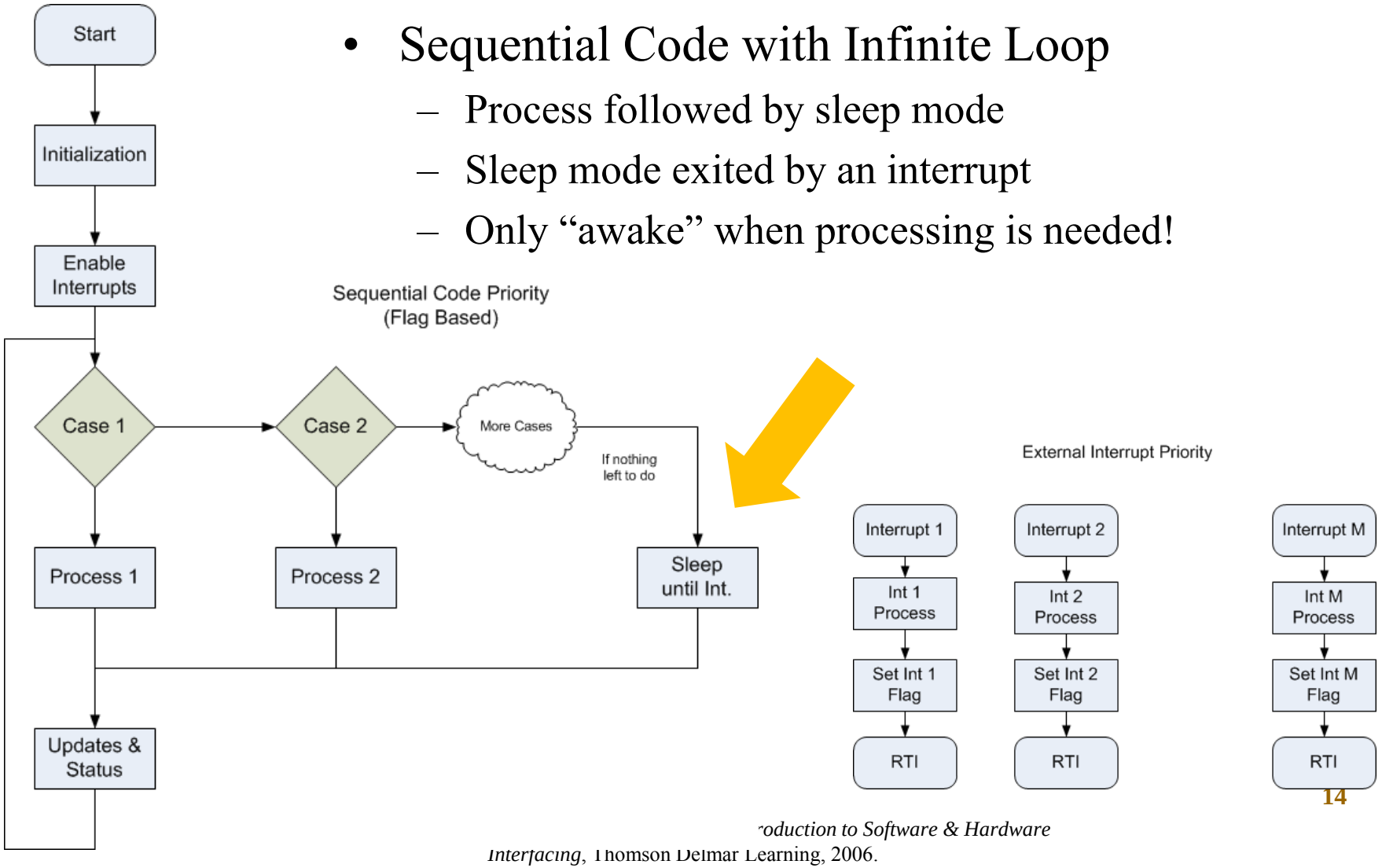
- Sequential Code with Infinite Loop
  - Process as time is available
- Interrupts that must be immediately processed



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

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!



roduction to Software & Hardware Interfacing, Thomson Delmar Learning, 2006.

Software Operating States

A: Initialization

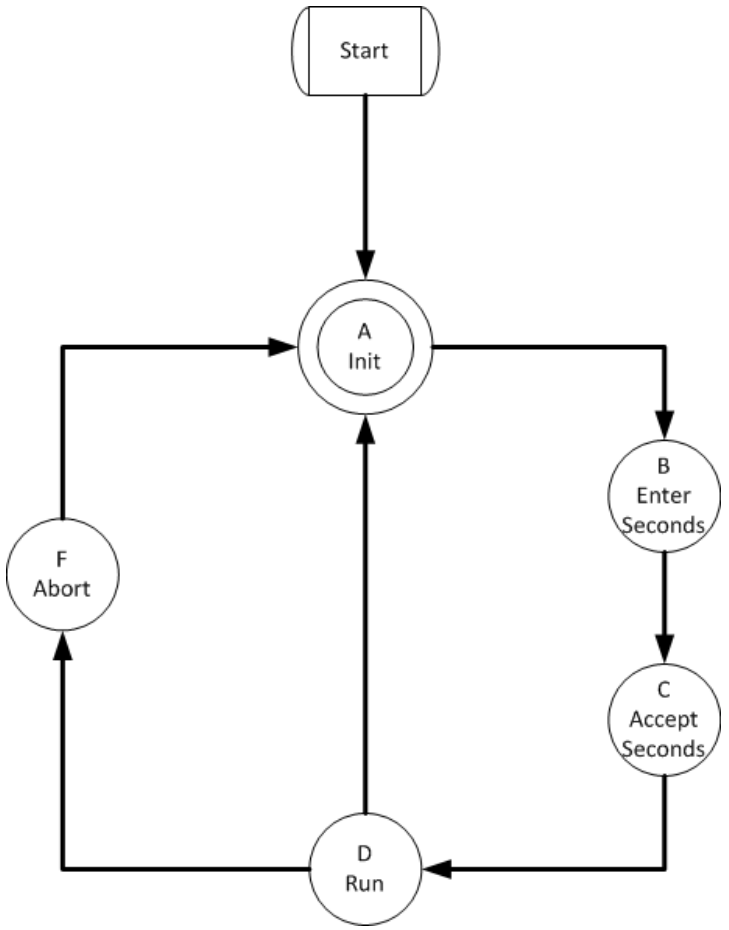
B: time in seconds data entry

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

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

Sunseeker Driver Controller "State" Update Code (1 of 2)

```
// 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;
```

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

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

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)

```
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 && brake){ dcMODE = FWD_RDY; }
        else if(cruise){
                dcMODE = CRCNTRL;
                switches_new &= ~(SW_REGEN);
                cruise_velocity = actual_velocity;
                cruise_current = avg_set_current;
                ruise_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 && 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;
}
// Mode update completed
```

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

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

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

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

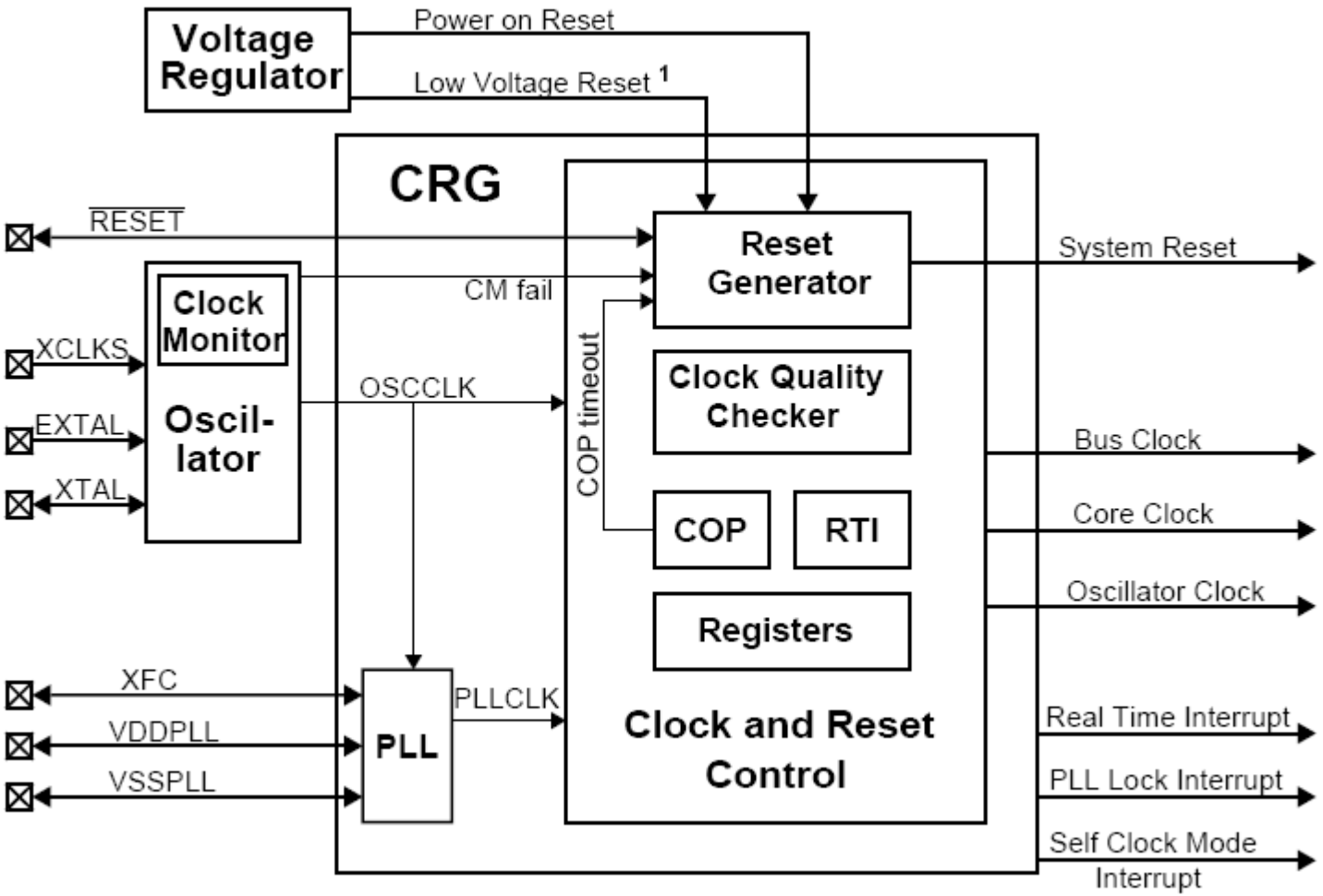
Alternate State Method 2

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

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

        }
        else if(state == "B")
        {

        }
        else
        {

        }

        … etc …



}
```

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

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

Clock and Reset Generation Block (CRG) (CRG_SW code setup)



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

Set Choice of Clock Sources

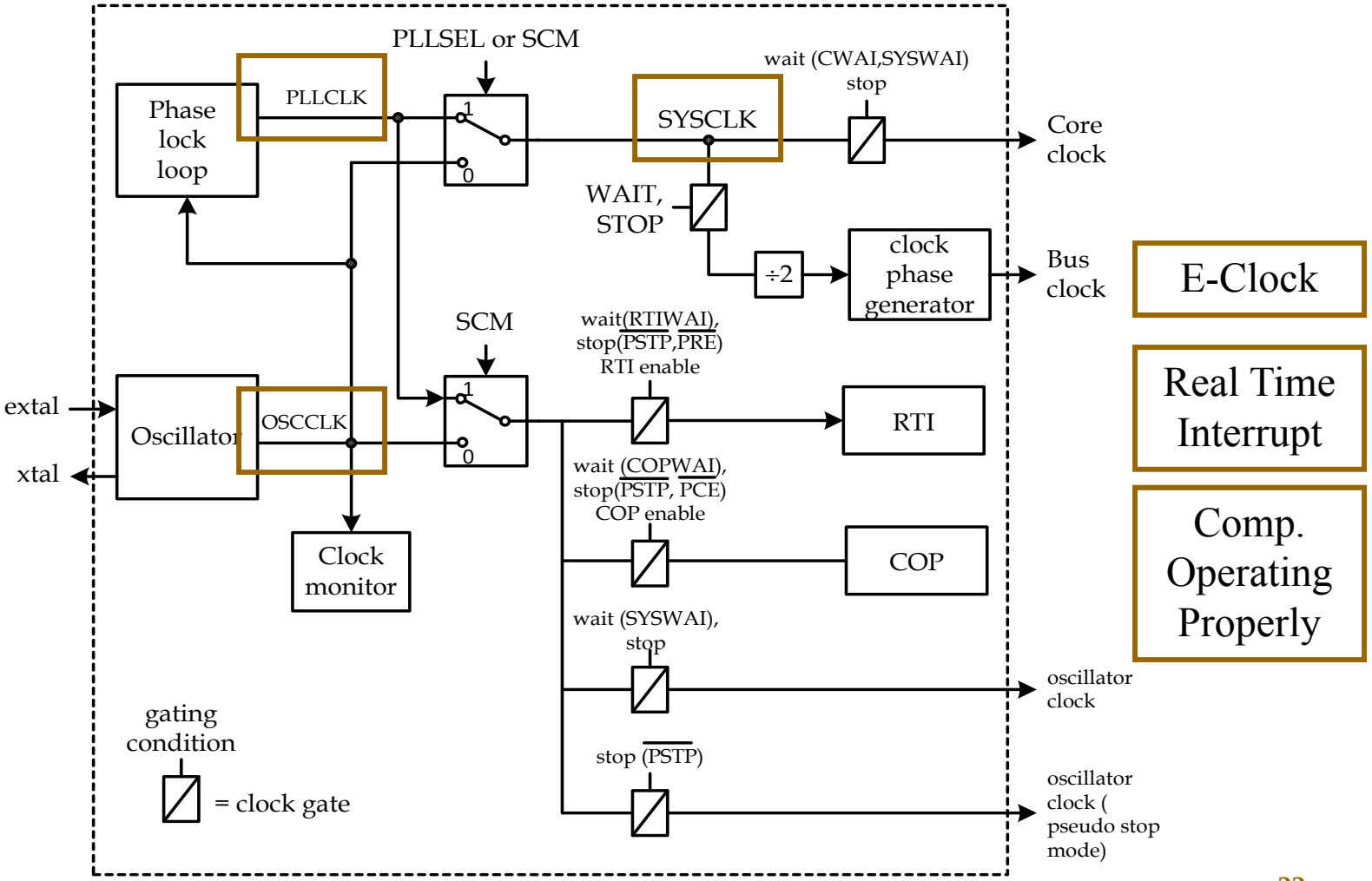


Figure 6.15 HCS12/9S12 clock generation circuit

Material from Figure 6.15 based on: *The 9S12/9S12X Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

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

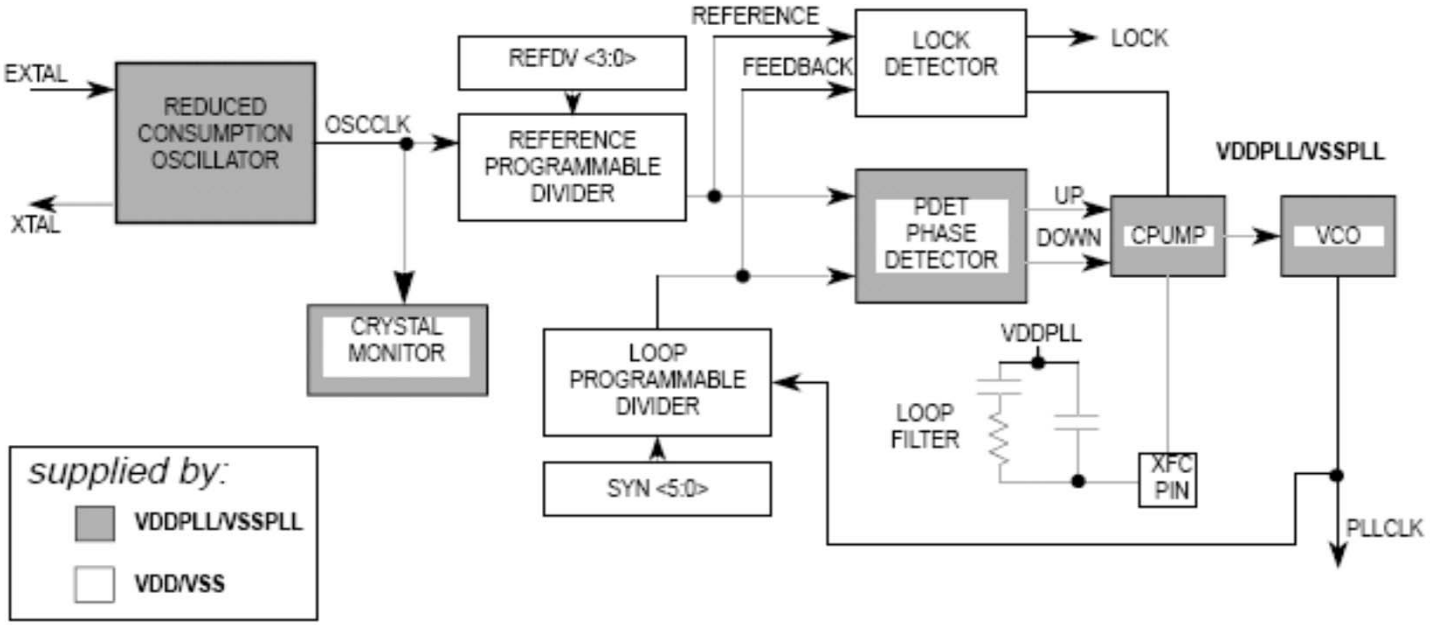


Figure 4-1 PLL Functional Diagram

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

CRG Block User Guide V02.07, Doc. Number S12CRGV2/D

CRG: Clock and PLL Setting

CRG_sw.h

```
// Public Function Prototypes
void pll_init(void);
void crg_init(void);
void rti_isr(void);

// Defines for ECLK access and PLL setup
#ifndef __ECLK_ACCESS
#define __ECLK_ACCESS
#endif

#ifndef __SET_PLL
#define __SET_PLL
#endif

// PLL and ECLK Rate Setting
#define BUSRATE                                 24
#define XTALRATE                16  //16 MHz Xtal,
effective 16 MHz OSCCLK
#define PLLCOMPRATE             1   // Note
max(XTAL/PLLC) = 16

// Module LED pin access
#define LED_PORT PTP
#define LED_PIN PTP7

// Constant Definitions
#define             TRUE                        1
#define FALSE                   0
```

```
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
  SYNR  = (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

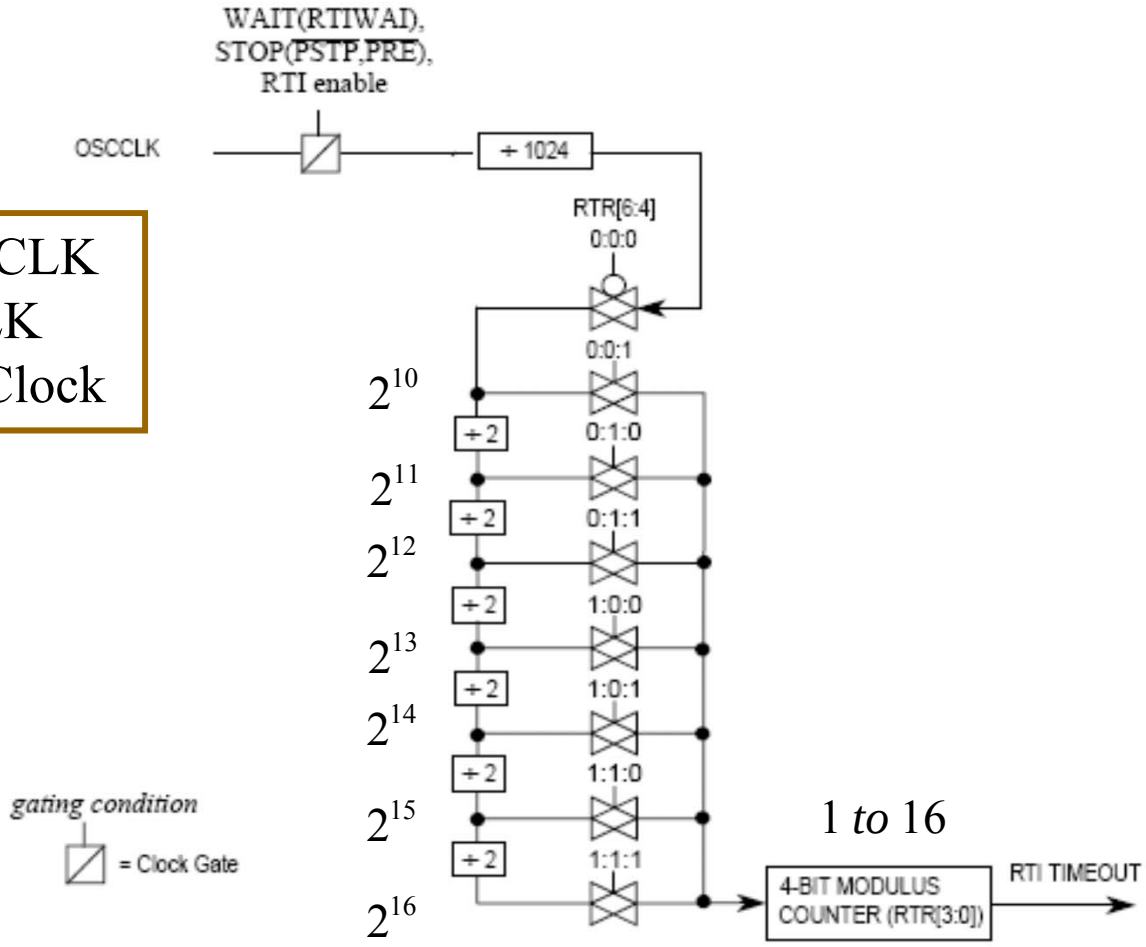


Figure 4-6 Clock Chain for RTI

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

CRG Block User Guide V02.07, Doc. Number S12CRGV2/D

CRG: RTI Initialization and ISR

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

#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;
  }
}
```

```
#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

#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
```

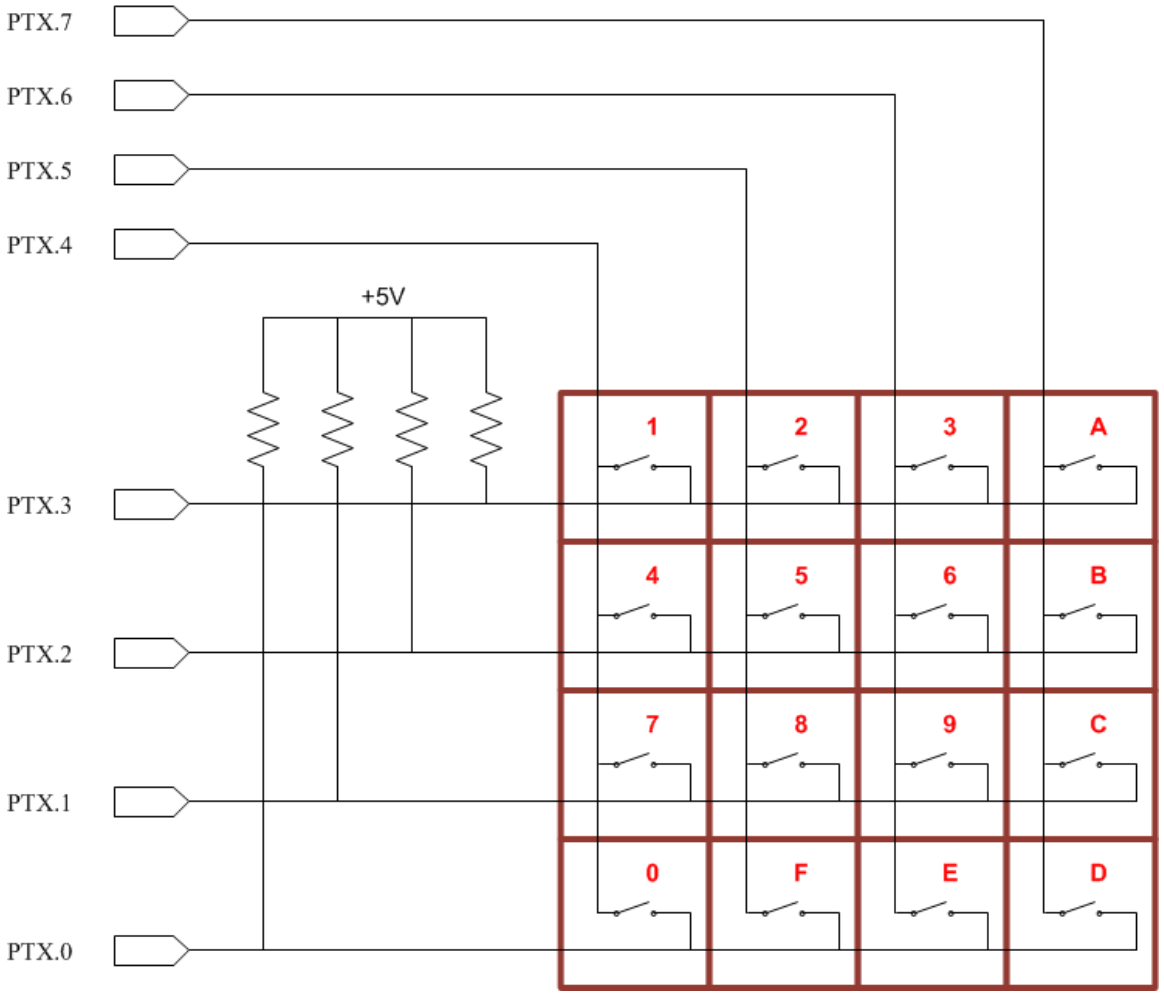

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Parallel Pin Init:
SPI Port Init (for SS Pin)

```
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 buad 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 |= SPISWAI;             //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
}
```


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

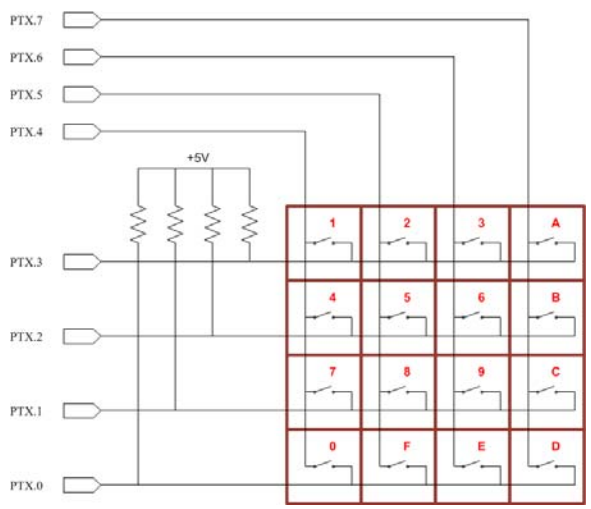
Class Keypads



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

A GetKey function

```
# 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;
                keypad  = cmask;                // Test the 0th row
                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 0x 7F
        }
        keypad = 0xFF;                          // standby keypad values
        return (0xFF);
}
```



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

Main Code: Periodic Keypad Test

```
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) {      // accept if greater
                                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;
                }
                last_key = current_key;
}       }       }
```

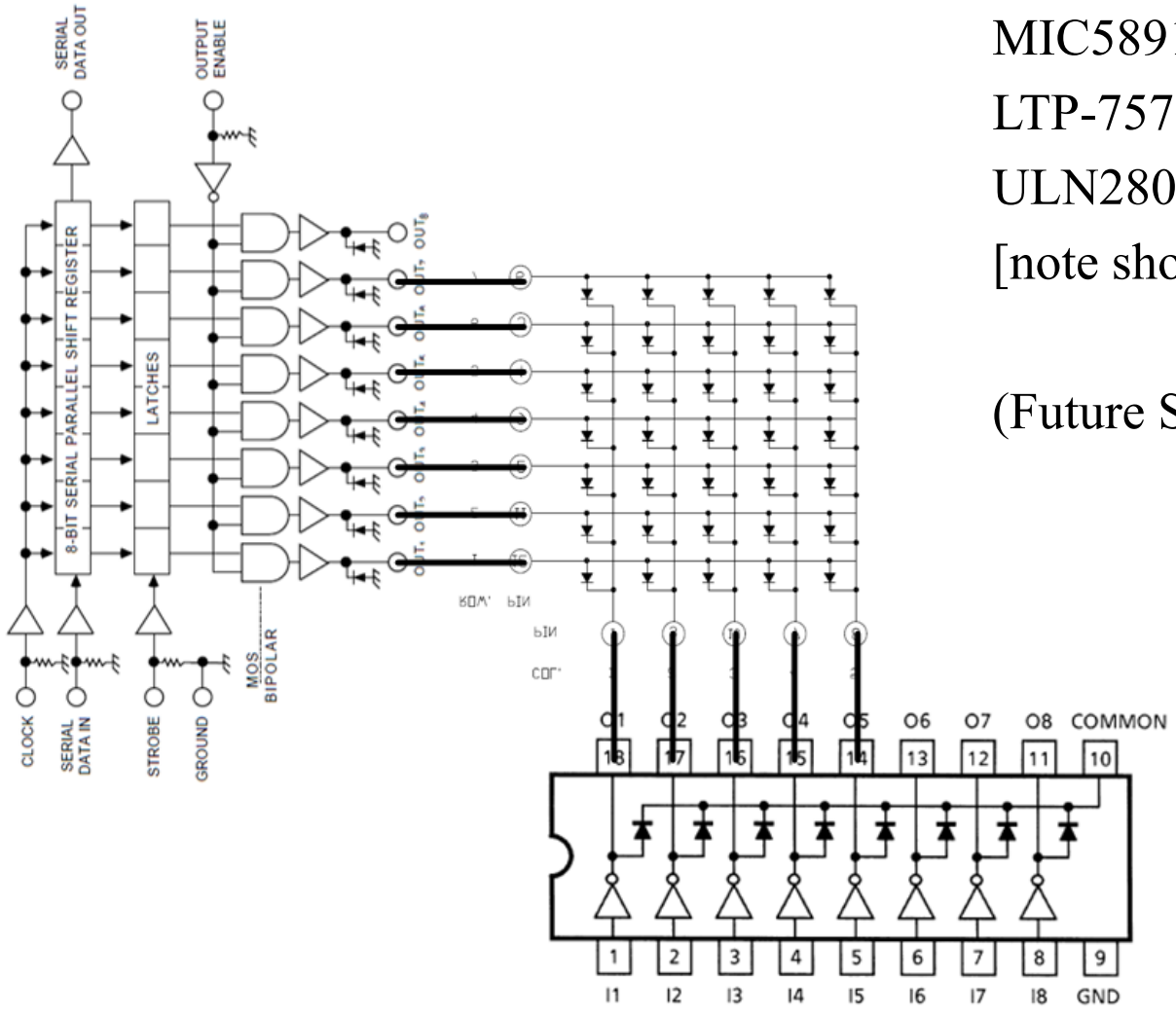
If key_status=0x03, do something about the value written!

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

Keypad Main Code Misc

```
# define        keypad          PORTA
# define        keypad_dir      DDRA
# define        KEY_THRESH 1000

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");

// Initilize 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++)           // initilize key_input
{
        key_input[ii]=0;
}

asm("cli");
```

5x7 Matrix with Source and Sink ICs



MIC5891 Source

LTP-757G or similar Display

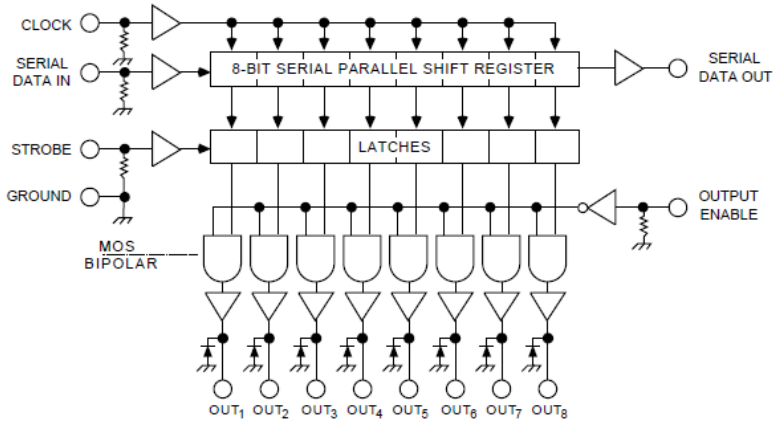
ULN2803A Sink

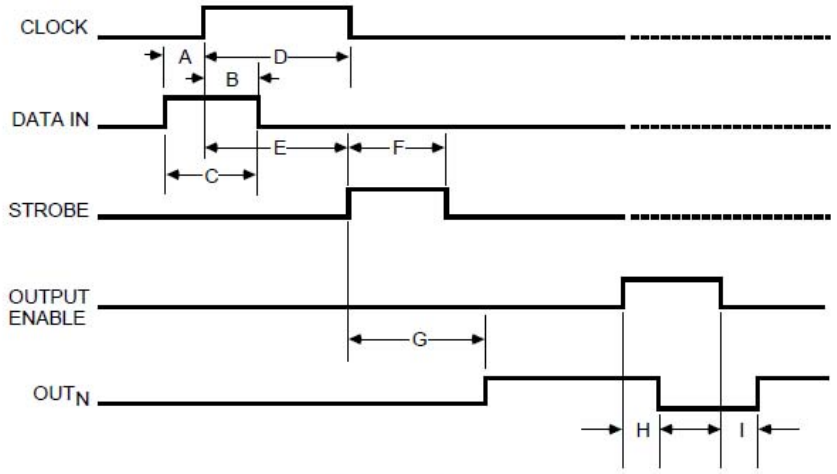
[note shown 74HCT595 shift reg.]

(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





Timing Conditions

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

Bit Banging MIC row data

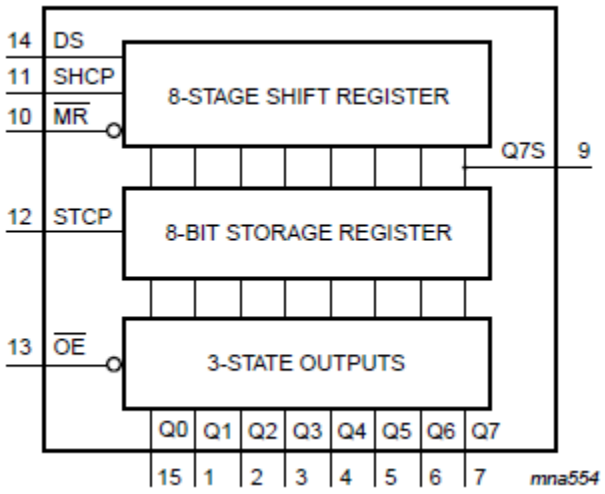
- Using software and individual bit-level ports to create a more complex signaling stream (PTP.0 data, PTP.1 clock)

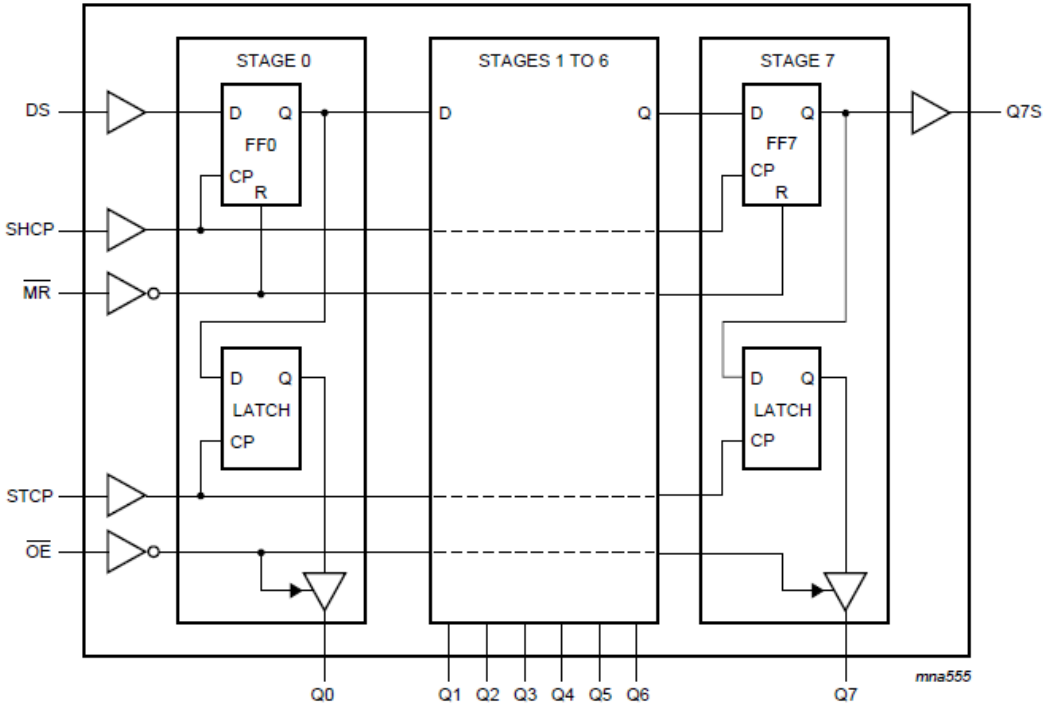
```
row_load(rows_excited[jj]);
void row_load(char temp){
int ii;
for (ii=0;ii<8;ii++) {
    bit_value = temp & 0x01;              \\ Determine value
    if(bit_value == 0x01) PTP |= 0x01;    \\ Data output
    else PTP &= ~(0x01);
    asm("nop");                           \\ setup > 75ns
    asm("nop");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


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

Logic for Selecting Columns

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



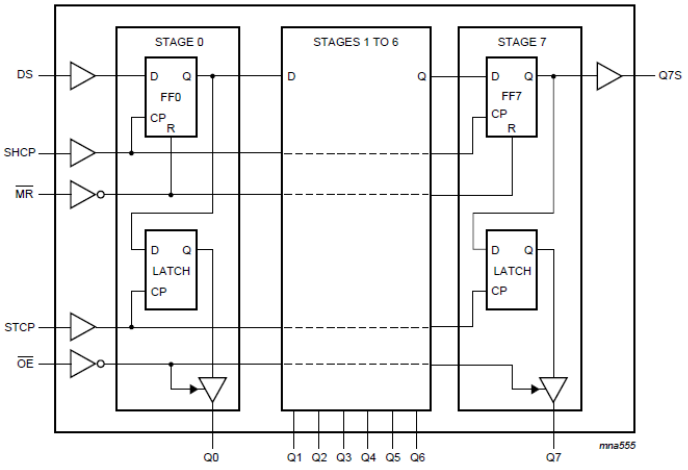


Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Functional Table

Table 3. Function table[1]

Control				Input	Output		Function
SHCP	STCP	$\overline{OE}$	$\overline{MR}$	DS	Q7S	Qn	
X	X	L	L	X	L	NC	a LOW-level on $\overline{MR}$ only affects the shift registers
X	↑	L	L	X	L	L	empty shift register loaded into storage register
X	X	H	L	X	L	Z	shift register clear; parallel outputs in high-impedance OFF-state
↑	X	L	H	H	Q6S	NC	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 Q6S) appears on the serial output (Q7S).
X	↑	L	H	X	NC	QnS	contents of shift register stages (internal QnS) are transferred to the storage register and parallel output stages
↑	↑	L	H	X	Q6S	QnS	contents of shift register shifted through; previous contents of the shift register is transferred to the storage register and the parallel output stages



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
ar Learning, 2006.

Bit Banging column shift register

- Using two cascaded 74HCT595

```
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
```


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

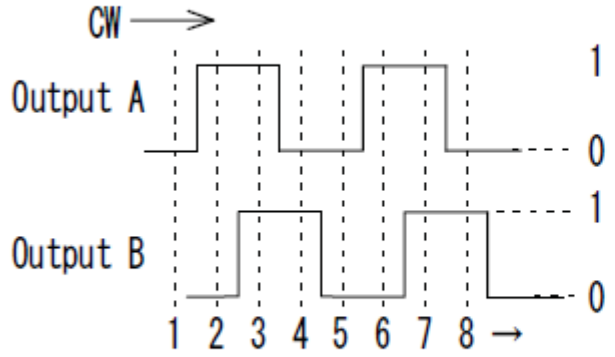
5x7 Display Code Function

```
if(display5x7_advance_flag)                                          // 1 cycle load column bit
{                                                                    // 15 cycles load row data shift column bit and enable
        display5x7_advance_flag = FALSE;                             // 1 cycle shift column bit to 16th/empty column
        PORTA |= 0x08;                  //  Output disable row source // Operational rate approx. 60 Hz * 17 ~ 1020 Hz
        switch(display_count)                                        // Using rti @ 1.024 msec => 976 Hz
        {                                                            // or 57.4 complete scans/sec
        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
                                        // The next clock will cause the 1st column to be active
                display_count++;        // increment counter
                break;
        case 1:case 2:case 3:case 4:case 5:case 6:case 7:case 8:case 9:
        case 10:case 11:case 12:case 13:case 14:case 15:
                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
```


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

Optical Rotary Shaft Encoder

OUTPUT WAVEFORM



Output \ Position	1	2	3	4	5
A	0	1	1	0	0
B	0	0	1	1	0

＊"0":1V max. "1":3V min.

＊The code repeats from 1 to 4.

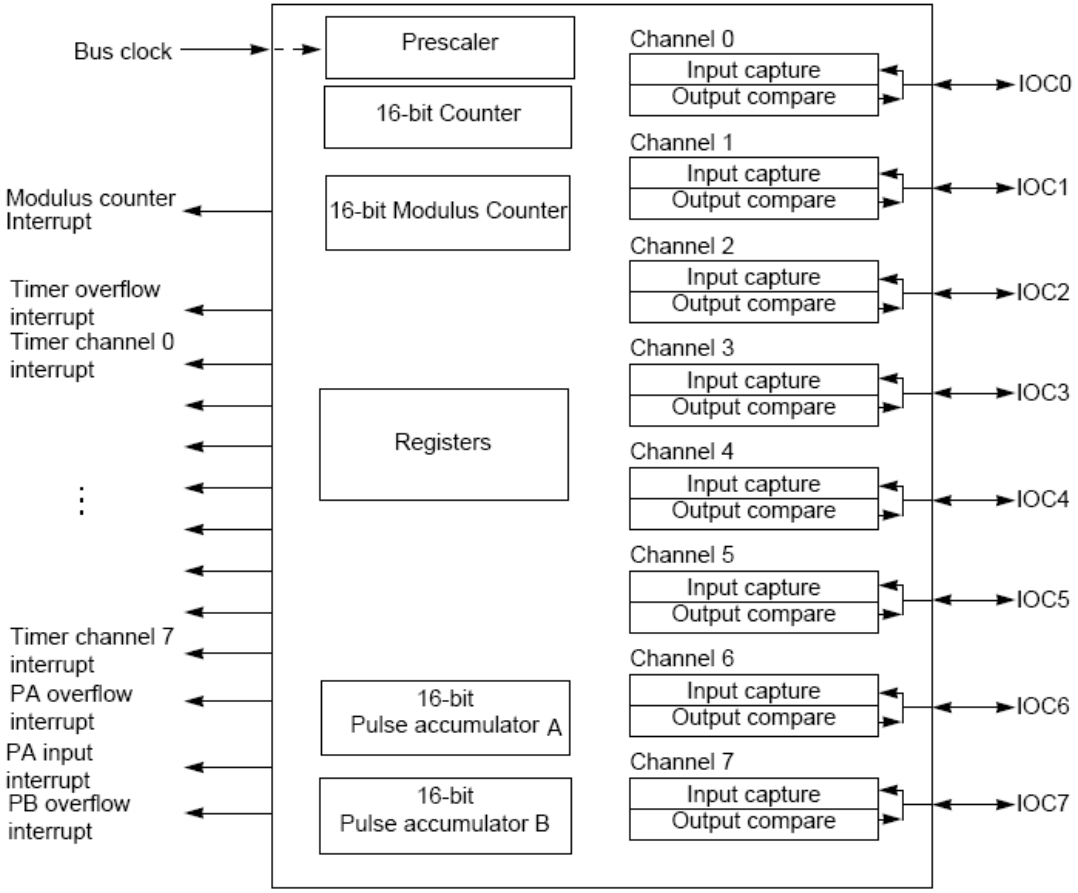
- Detect turn direction by which rising edge leads.
  - 16 or 32 position
  - 11.25 degree or 22.5 degree for code change
- Similar device used on Sunseeker to engage and control cruise control

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

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 void delayby1ms(int k)

```
/*************************************************************************/
/* The following function creates a time delay which is equal to the  */
/* multiple of 1 ms. The value passed in Y specifies the number of one*/
/* milliseconds to be delayed.                                        */
/*************************************************************************/
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

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

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

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

Delay Using ECT TCNT Counter

```
/* 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

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

Periodic Event Timing
Possible Code ?

```
void modcnt_init(void)
{
        // Modulus Down Counter Iniitalization
        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;
        }
}
```

FLAG_TIME must be defined in a header file

```
#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
```


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

Periodic Event Timing Usage (Try it and I will fix it if necessary)

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

```
#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");
}
```


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

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

```
#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 Code Pulse Width (2 of 2)

```
        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;
}
```


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

Example 8.4 Output Waveform

```
#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 */
    }
}
```


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

```
#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)

```
        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 ++;
}
```


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

C Program for Siren Generation (1 of 2)

```
#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");
```


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

C Program for Siren Generation (2 of 2)

```
        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
```


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

PWM Block Diagram

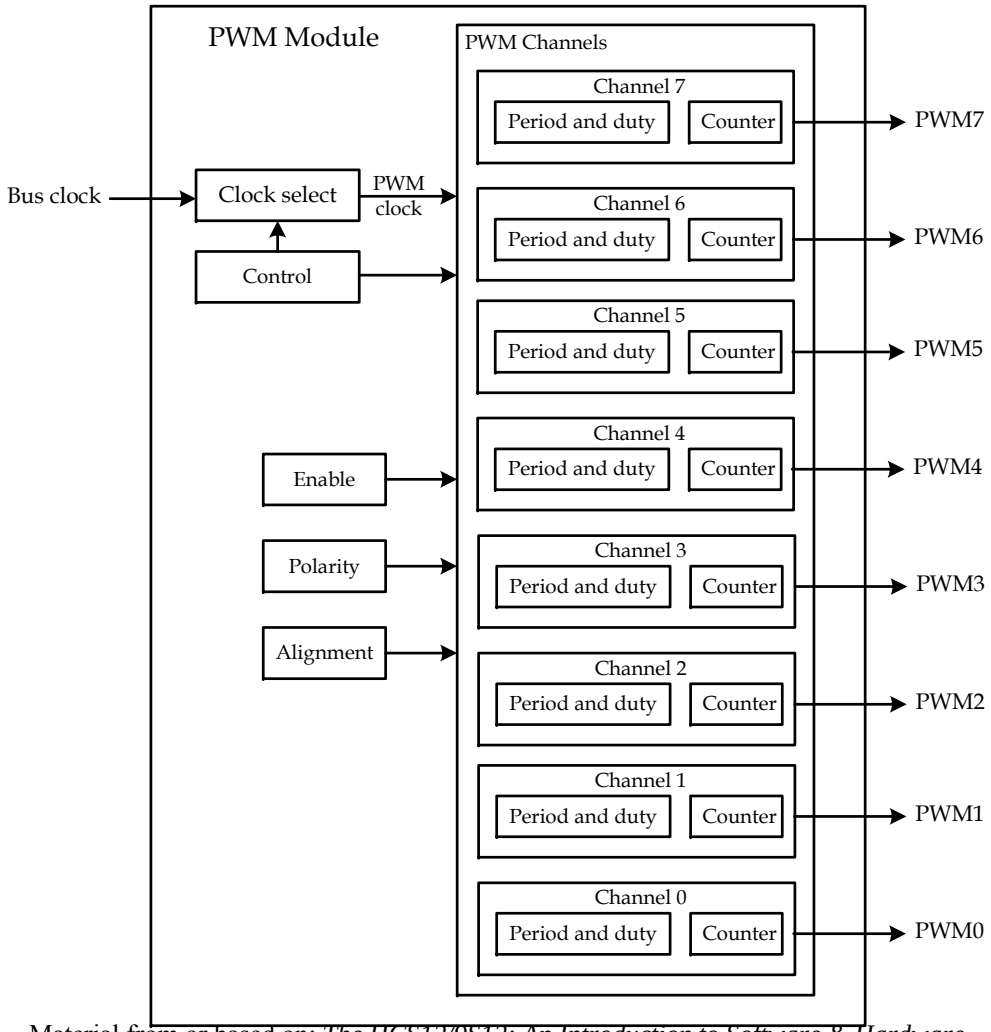


Figure 8.38 HCS12 PWM block diagram

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

PWM_Test.c (8-bit Reg/Cnt)

```
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++;
 }
```

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

 while(state == 4){
   asm("nop");
 }
}
```


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

void init_pwm(void)

```
void init_pwm(void)
{
  PWMPRCLK = 0x44;          // Presclae 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
}
```


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

PWM_Test2.c (16-bit Reg/Cnt)

```
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");
  }
}
```


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

PWM_Test2.c (16-bit Reg/Cnt)

```
void init_pwm(void)
{
  PWMPRCLK = 0x33;          // Presclae 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
}
```


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

The HCS12 SCI Subsystem (3 of 3)

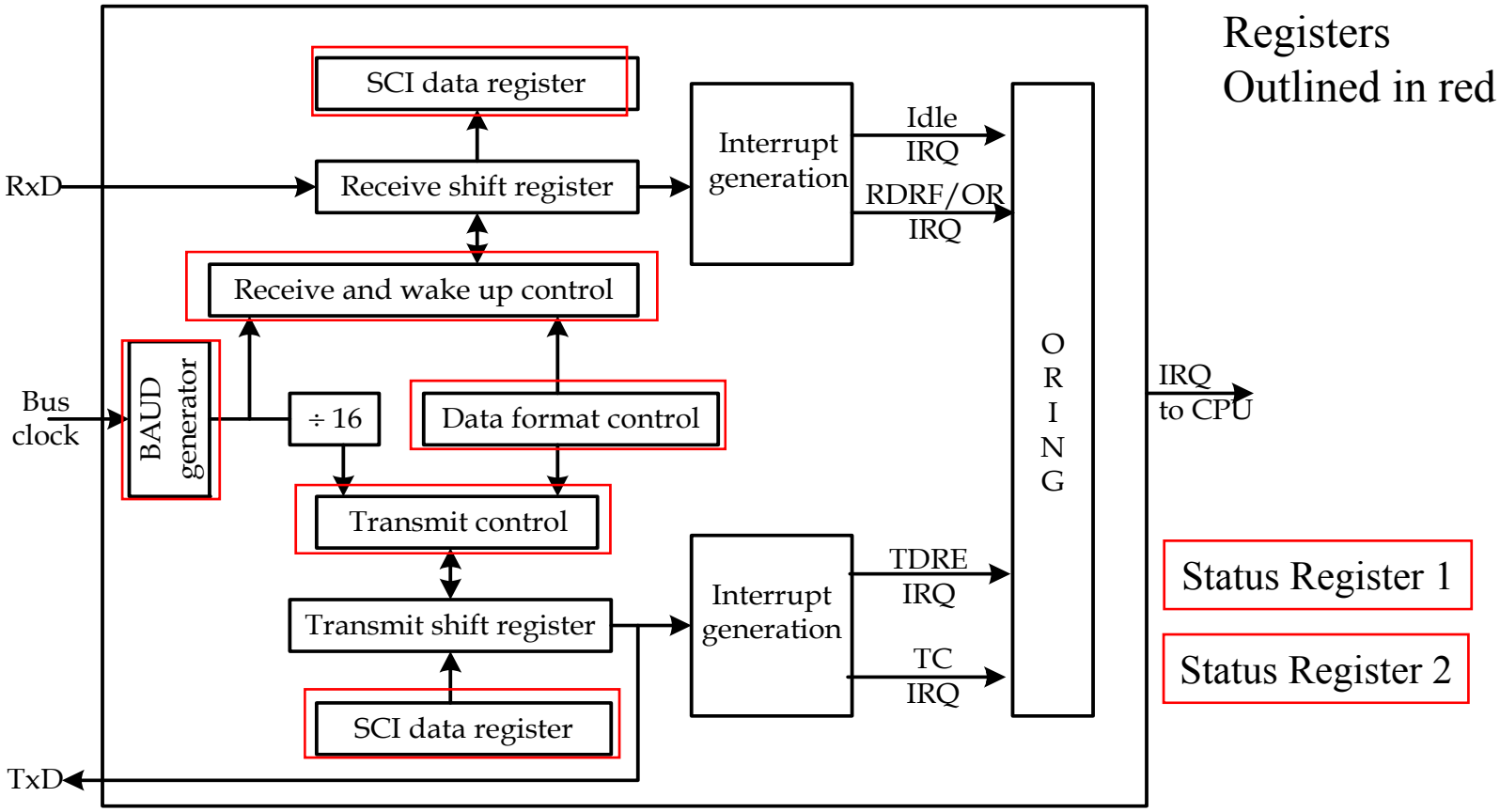


Figure 9.8 HCS12 SCI block diagram

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

SCI_Test_sw.c

```
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();
  }
}
```


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

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;

}

/*******************************/
/* The following function uses polling
    method to read a character */
/* from SCI0 port.                    */
/*******************************/
int getchar(void)
{
   while(!(SCI0SR1 & RDRF));
   return SCI0DRL;
}
```

```
/*************************************/
/* The following function uses the
  polling method to output a    */
/* character to the SCI0 port.                */
/*************************************/
int putchar(char cx)
{
   while(!(SCI0SR1 & TDRE))
     continue;
   SCI0DRL = cx;
   return 0;
}

/*************************************/
/* The following function uses
   polling method to read a character */
/* from SCI0 port.                            */
/*************************************/
char checkchar(void)
{
    if(SCI0SR1 & RDRF)
           { return SCI0DRL;  }
           else
           {  return 0;  }
}
```


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

SCI0 Functions

```
/*****************************************/
/* The following function reads a string from
   the SCI0 port  */
/*****************************************/
int getstr(char *ptr)
{
   while (1) {
        *ptr = getchar();
        if (*ptr == 0x0D){
          *ptr = 0;
          return 0;
        }
        else ptr++;
    }
   return 0;
}
/*****************************************/
/* The following function outputs a carriage return
   and a linefeed  */
/*****************************************/
int newline(void)
{
    putchar(0x0D);
    putchar(0x0A);
    return 0;
}
```

```
/*****************************************/
/* The following function copies a string backwards     */
/*****************************************/
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;
}
```


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

Output to a new line

```
/*****************************************************************/
/* 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;
}
```


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

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;
}
```


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

Using the stdio library

Input-output manipulation functions

stdin – makes a function call to getchar

stdout – makes a function call to putchar

Name	Notes
getc	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
getchar	has the same effects as getc(stdin)
gets	reads characters from stdin until a newline is encountered and stores them in its only argument
printf	**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)**

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

SS Multiple IC Interconnection

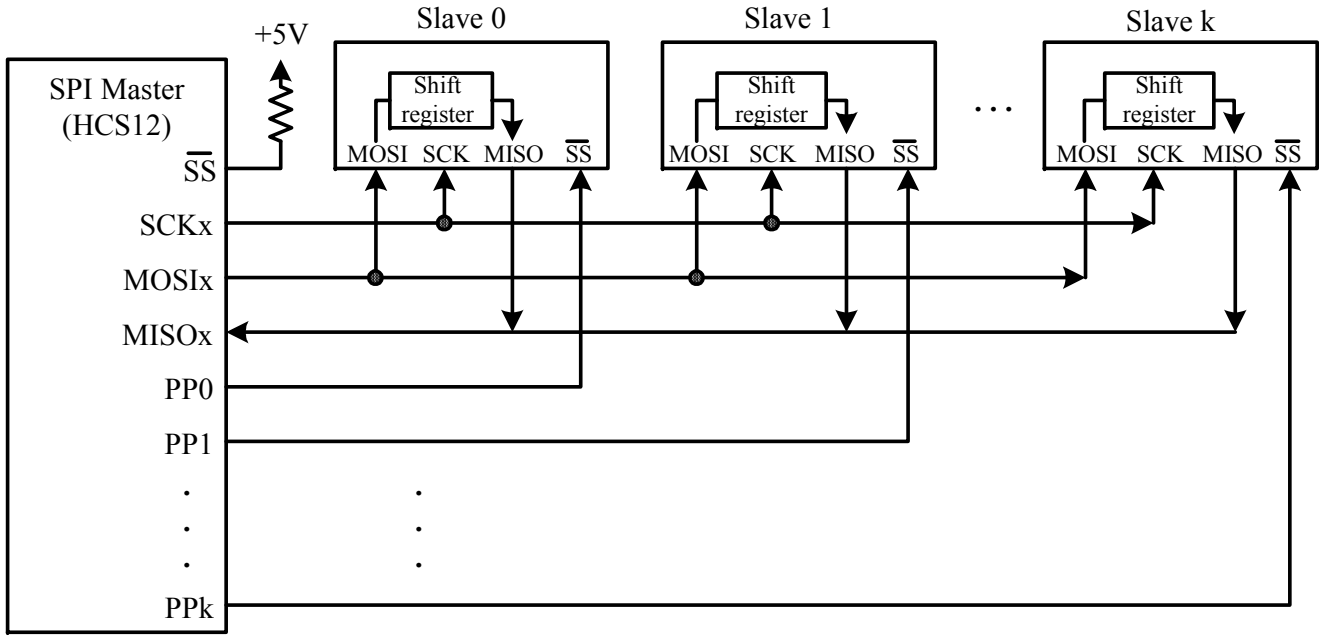


Figure 10.9 Single-master and multiple-slave device connection (method 1)

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

SPI_Test1
(MOSI to MISO loopback)

```
void main(void){

COPCTL=0x00;
asm("sei");
pll_init();
crg_init();
init_sci0();
init_spi0();
DDRH = 0x00;
DDRS |= 0x80;
//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");
  }
}

printf(data_buffer1);
```

```
// 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=exchcspi0(*tx0_ptr);
     tx0_ptr++;
     rcv0_ptr++;
    spi0_deselect;
    delayby1us(1);
  }

  *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()
```


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

void init_spi0(void)

```
/*****************************************************************************/
/* Similar to Example 10.3 1st ed. initialization routine                    */
/*****************************************************************************/
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;     // Make a parallel pin. Explicit sw operation.
// SPI0 port interface macros
#define spi0_select                 PTS &= ~PTS7;
#define spi0_deselect               PTS |= PTS7;

  SPI0BR = 0x20;//set buad 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 |= SPISWAI;            //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 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)

```
/*****************************************************************************/
/* The following function exchanges a character with the SPI0 interface.    */
/*****************************************************************************/
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 */
}
```


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

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 */
}
```


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

SPI_Test2
(spi0 master spi1 slave)

```
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);
```

```
// Designed to test SPI0-Master to SPI1-Slave
// Note: this only works for 8-bit transfers
while(state == 3){

  tx0_ptr = &data_buffer1[0];
  rcv0_ptr = &in_buffer1[0];
  tx1_ptr = &data_buffer2[0];
  rcv1_ptr = &in_buffer2[0];

// Transfer the data
  while (!(*tx0_ptr==0)){
    putcspi1(*tx1_ptr);              //Slave write
    tx1_ptr++;

    spi0_select;
    *rcv0_ptr=exchcspi0(*tx0_ptr);  //Master exchange
    tx0_ptr++;
    rcv0_ptr++;
    spi0_deselect;

    *rcv1_ptr=getcspi1();            // Slave read
    rcv1_ptr++;

    delayby1us(1);
    }
```

```
// Print the master and slave input buffers
  *rcv0_ptr = 0;
  *rcv1_ptr = 0;
  printf(in_buffer2);
  newline();
  printf(in_buffer1);
  newline();

// Clear the master and slave input buffers
  rcv0_ptr = &in_buffer1[0];
  rcv1_ptr = &in_buffer2[0];
  for(ii=0;ii<80;ii++){
    *rcv0_ptr=0;
    rcv0_ptr++;
    *rcv1_ptr=0;
    rcv1_ptr++;
    }

  delayby1us(5);
  }
}  // end of main()
```


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

void init_spi1(void)

```
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 buad rate to 24 MHz/6 = 4 MHz
  SPI1CR1 |= SPE | CPHA; //Enable, Master, SCLK high active low, even phases
  SPI1CR1 &= ~(SSOE | 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
}
```


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

putcspi1 and getcspi1

```
/*****************************************************************************/
/* The following function outputs a character to the SPI1 interface.         */
/*****************************************************************************/
void putcspi1 (char cx)
{   char temp;
    while(!(SPI1SR & SPTEF));          /* wait until write is permissible */
    SPI1DR = cx;                       /* output the byte to the SPI */
}

/*****************************************************************************/
/* The following function reads a character from the SPI1 interface.         */
/*****************************************************************************/
char getcspi1(void)
{
    while(!(SPI1SR & SPIF));           /* wait until a byte has been shifted in */
    return SPI1DR;                     /* return the character and clear SPIF flag */
}
```


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

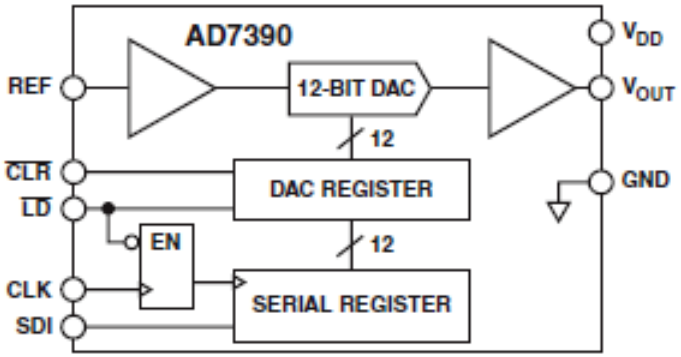
DAC AD7390

ANALOG DEVICES

3 V Serial-Input
Micropower 10-Bit and 12-Bit DACs

AD7390/AD7391

FUNCTIONAL BLOCK DIAGRAM



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

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

Example DAC AD7390

```
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
}
```


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

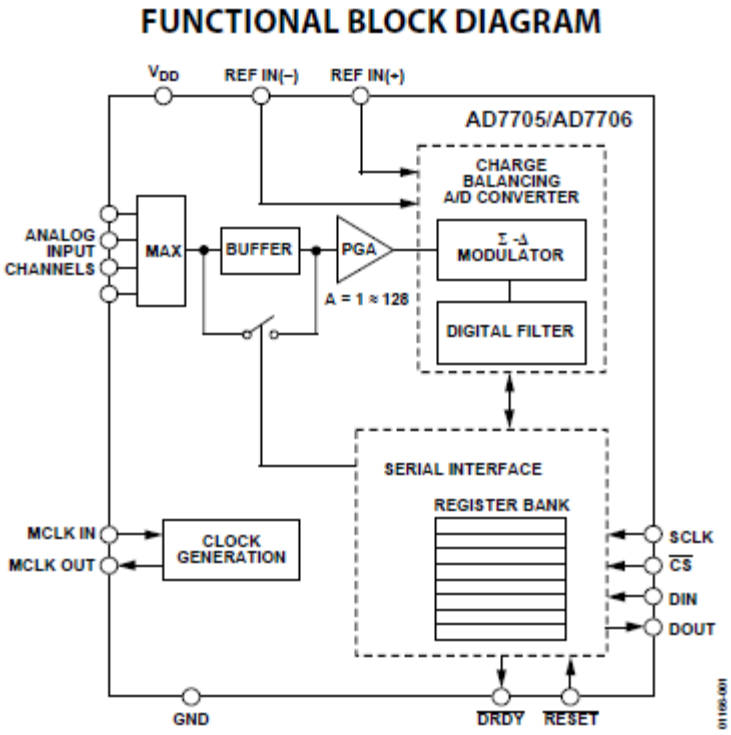


Figure 1.

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

AD7705.c elements

```
#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;
}
```


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

AD7705.c elements

```
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;
}
```


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

AD7705.c elements

```
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

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

#define COMM_REG    0x00;
#define SETUP_REG   0x10;
#define CLOCK_REG   0x20;
#define DATA_REG    0x30;
#define CAL_REG     0x60;
#define GAIN_REG    0x70;

#define READ_REG    0x08;
#define WRITE_REG   0x00;

#define STANDBY     0x04;

#define CH0         0x00;
#define CH1         0x01;
#define CHGNDREF    0x02;
#define CHGNDS      0x03;
```

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

```
void adc_setup_norm(void)    //setupthe ADC
{
        adc_select;
        adcspi_transmit(SETUP_REG|WRITE_REG|CH0);
        adcspi_transmit(GAIN0);
        adc_deselect;
}
```


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

AD7705.h elements

```
#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

```
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

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

AD7705.h elements

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

#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;
}
```


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

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

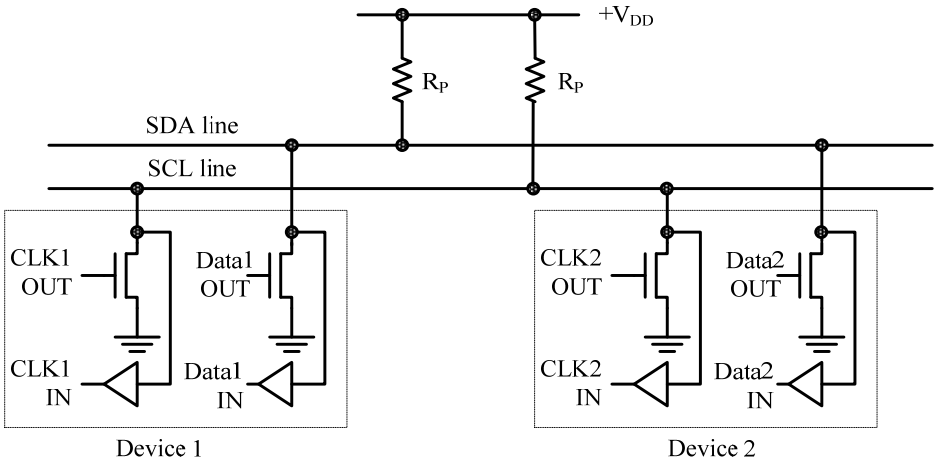


Figure 11.1 Connecting standard- and fast-mode devices to the I^2C bus

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

Byte Level Transfer

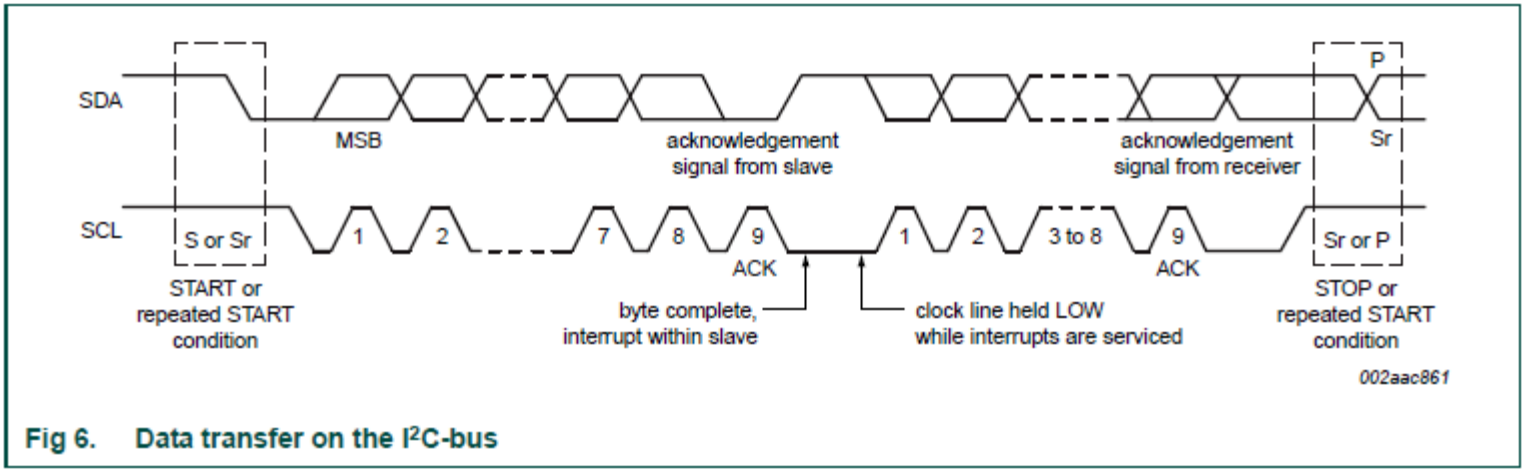


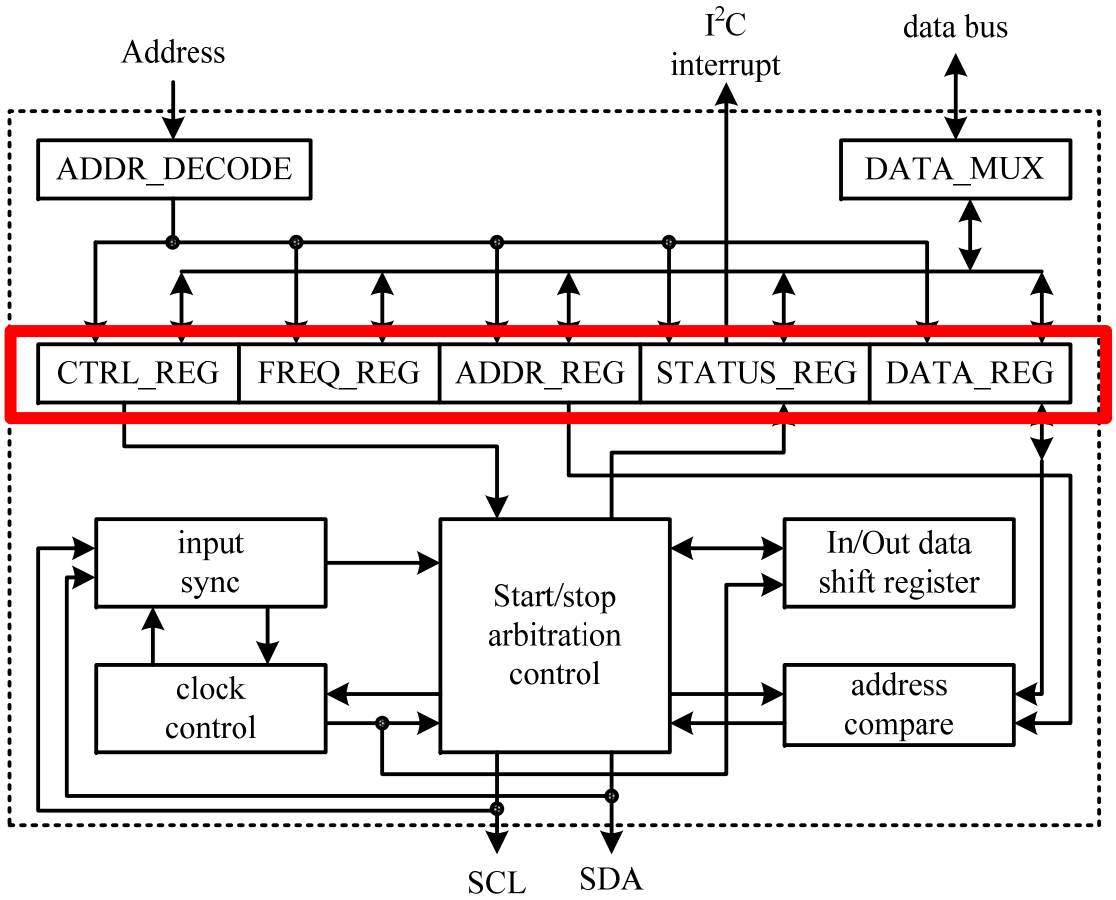
Fig 6. Data transfer on the I²C-bus

- 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
http://www.nxp.com/documents/user_manual/UM10204.pdf

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

IIC Block Diagram



Registers

Figure 11.21 I²C 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

```
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 */
}
```


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

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

```
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 */
}
```


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

I2C Master Sending Byte

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

```
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 */
}
```


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

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

```
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 □A 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

FIGURE 1-1: TYPICAL OPERATING CIRCUIT

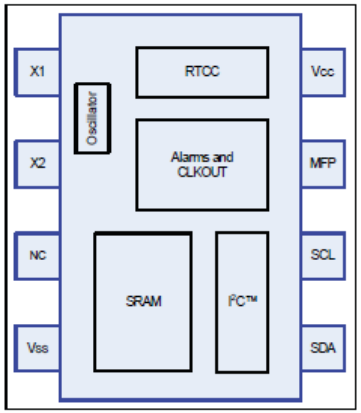


FIGURE 2-1: DEVICE PINOUTS

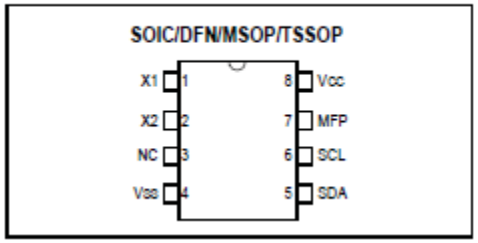


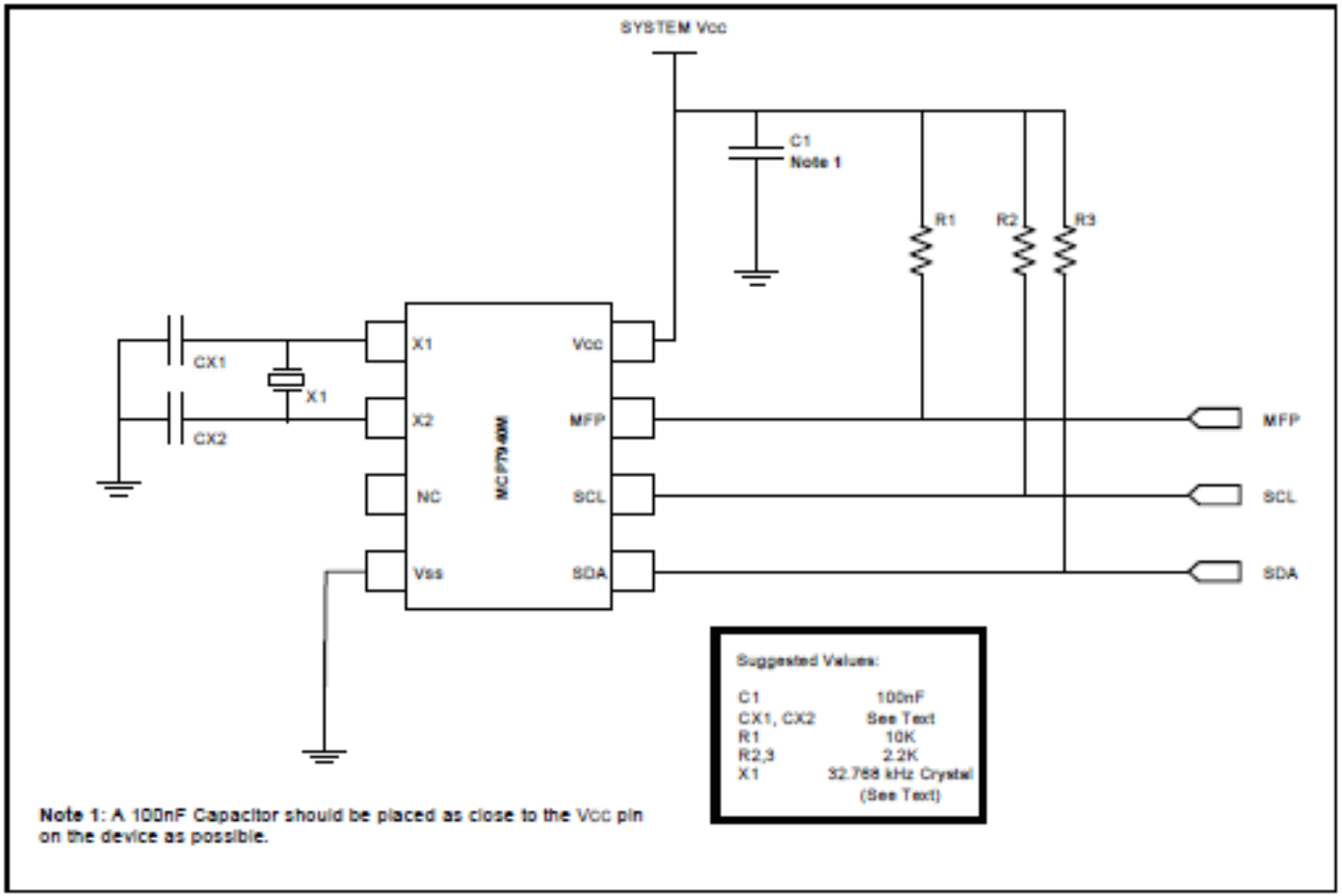
TABLE 2-1: PIN DESCRIPTIONS

Pin Name	Pin Function
VSS	Ground
SDA	Bidirectional Serial Data
SCL	Serial Clock
X1	Xtal Input, External Oscillator Input
X2	Xtal Output
NC	Not Connected
MFP	Multi-Function Pin
VCC	+1.8V to +5.5V Power Supply

Material from or based on: *The HCS12/9S12: An Introduction to Sof Interfacing*, Thomson Delmar Learning, 2006.

Typical Application Schematic

FIGURE 1-2: 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

CAN Physical Layer

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

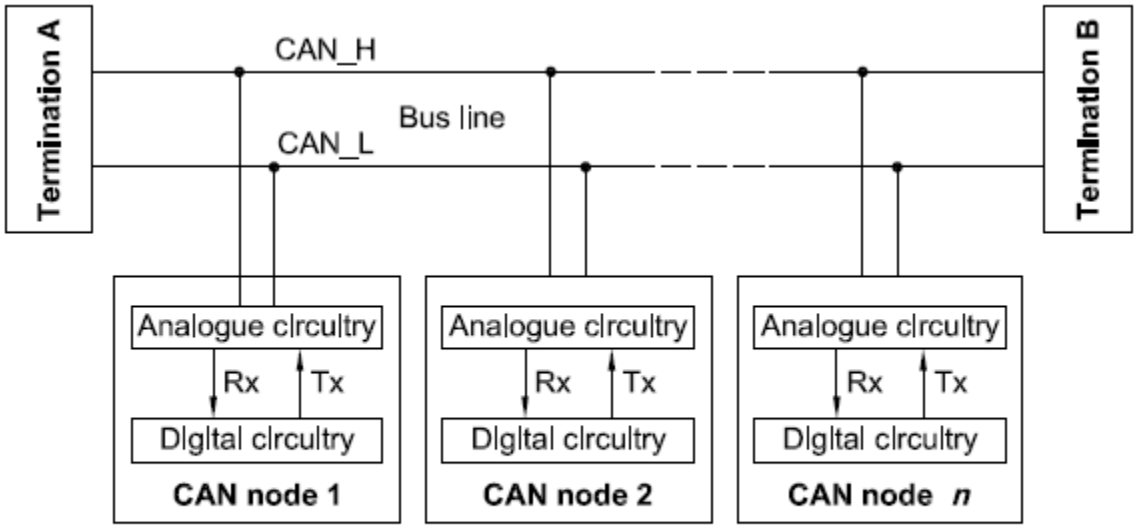


Figure 1 — Suggested electrical interconnection

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

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).

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

CAN_ECE2

```
main()
{
 unsigned char ii;

 COPCTL=0x00;
 asm ("sei");       //global interrupt disable
 pll_init();
 crg_init();
 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;
    }
   }

  }

 return(0);
}  // end of main()
```


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

void can0_init(void)

```
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);  //synchronizaiton 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
//    CAN0BTR0 = 0x43;             //Class default
//    CAN0BTR1 = 0x23;             //Class default

  CAN0IDAC |= IDAM0;               //four 16-bit acceptance filters

    CAN0IDAR0 = 0x5F;              //receive 11-bit address is 0x5FE0 -> 0x02FF
    CAN0IDAR1 = 0xE7;
    CAN0IDAR2 = 0x5F;              //receive 11-bit address is 0x5FE0 -> 0x02FF
    CAN0IDAR3 = 0xE7;
    CAN0IDAR4 = 0x5F;              //receive 11-bit address is 0x5FE0 -> 0x02FF
    CAN0IDAR5 = 0xE7;
    CAN0IDAR6 = 0x5F;              //receive 11-bit address is 0x5FE0 -> 0x02FF
    CAN0IDAR7 = 0xE7;
    CAN0IDMR0 = 0x00;              //11-bit address masks
    CAN0IDMR1 = 0x1F;
    CAN0IDMR2 = 0x00;
    CAN0IDMR3 = 0x1F;
    CAN0IDMR4 = 0x00;
    CAN0IDMR5 = 0x1F;
    CAN0IDMR6 = 0x00;
    CAN0IDMR7 = 0x1F;

    CAN0CTL0 &= ~(INITRQ);         //turn-off the initialization state

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

  //  CAN0TIER = TXEIE0;           //enable the transmit interrupt buffer 0
    CAN0RIER = RXFIE;              //enable the receive interrupt
    CAN0TBSEL = TX0;               //select the transmit buffer 0

  }
```


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

void can0_tx_start(void)

```
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
}
```


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

can0_tx_isr

```
#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
}
```


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

can0_rcv_isr

```
#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
}
```


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

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.

Using multiple transmit buffers (1)

```
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
```

Check for existing address

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

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
```

Check for free address

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

Using multiple transmit buffers (3)

```
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

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

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

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

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)

```
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

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

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);
}
```

Check RXF (RTR or message)

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

Using receive buffers (3)

```
        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).

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