68HC11 Notes

Version 1.1.1
Oct 30, 2006

Andrew J. Blauch
School of Engineering
Grand Valley State University

DISCLAIMER:
All software is provided “as is” and without any express or implied warranties, including, without limitation, the implied warranties of merchantability and fitness for a particular purpose.

COMPATABILITY:
This software has been created and tested using the following development systems:

Compiler(s):
GCC 68HC11 compiler version 2.2

Processor(s):
Motorola 68HC11 E1,E9 operating at 2 MHz E-clock

Evaluation Board(s):
Axiom Manufacturing CMM11E1-EVBU
Axiom Manufacturing CME11E9-EVBU

Simulators(s):
68HC11 EVBU Simulator Version 0.6. PySim11 Version 0.5
(developed by Andrew Sterian)

Table of Contents

List of Figures

List of Tables

1 HC11 Microcontroller

1.1 Overview

- Essential components packaged onto one chip
    - CPU (Motorola HC11)
    - Memory (internal)
    - I/O Peripherals (digital inputs/outputs, timers, etc)
- I/O Ports
    - External I/O signals grouped into 8-bit ports
    - There are five I/O ports (PORTA through PORTE)
    - Each port can be used as digital inputs/outputs
    - Each port also has additional functionality associated with it
        - PORTA – programmable timers
        - PORTD – serial interface
        - PORTE – analog-to-digital converter
        - PORTB/PORTC – expanded mode (address and data bus)
- There are different chip versions of the 68HC11
    - All based on the Motorola HC11 microprocessor
    - Transparent operation of I/O ports and registers
    - Versions vary in amount and type of memory and certain I/O features
- There are four operating modes
    - Operating mode determined by MODA/MODB pins on power-up
        - Single chip mode – All I/O ports available, external memory not available
        - Expanded chip mode – Not all I/O ports available, external memory available
        - Test modes – Used for factory testing and software installation
- The EVBU is an evaluation board with the microcontroller, serial hardware, external memory, and connectors. The EVBU provides an platform for quick development and testing of applications microcontroller applications

1.2 Memory Maps

- All hardware accessed by software via addresses
- Each hardware devices is mapped to a certain range of addresses
- Available hardware devices
    - Internal memory
    - External memory
    - I/O registers
- Additional hardware devices can be connected and mapped to the microcontroller using address decoding.

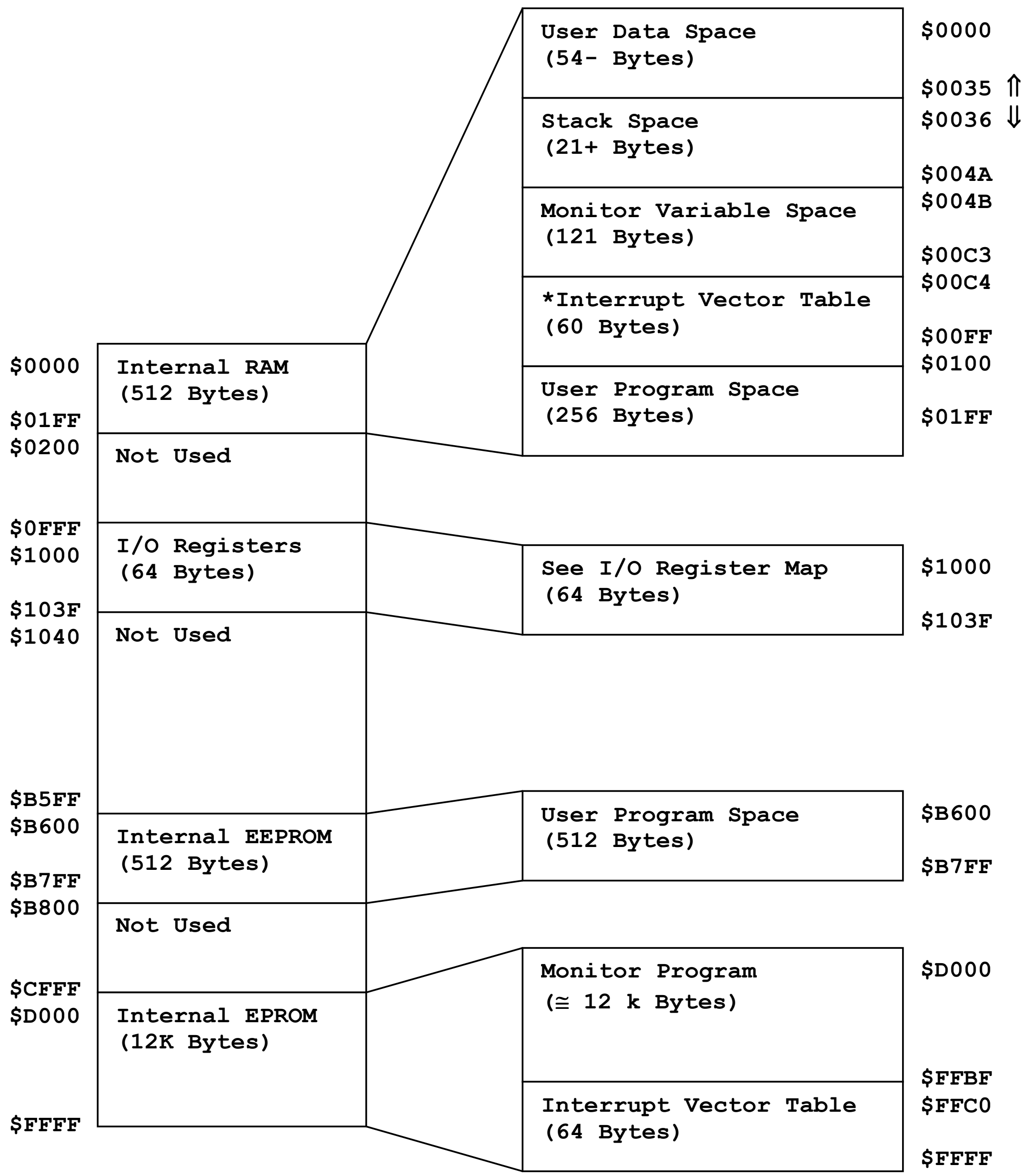


Figure 1: 68HC11E9 Memory Map

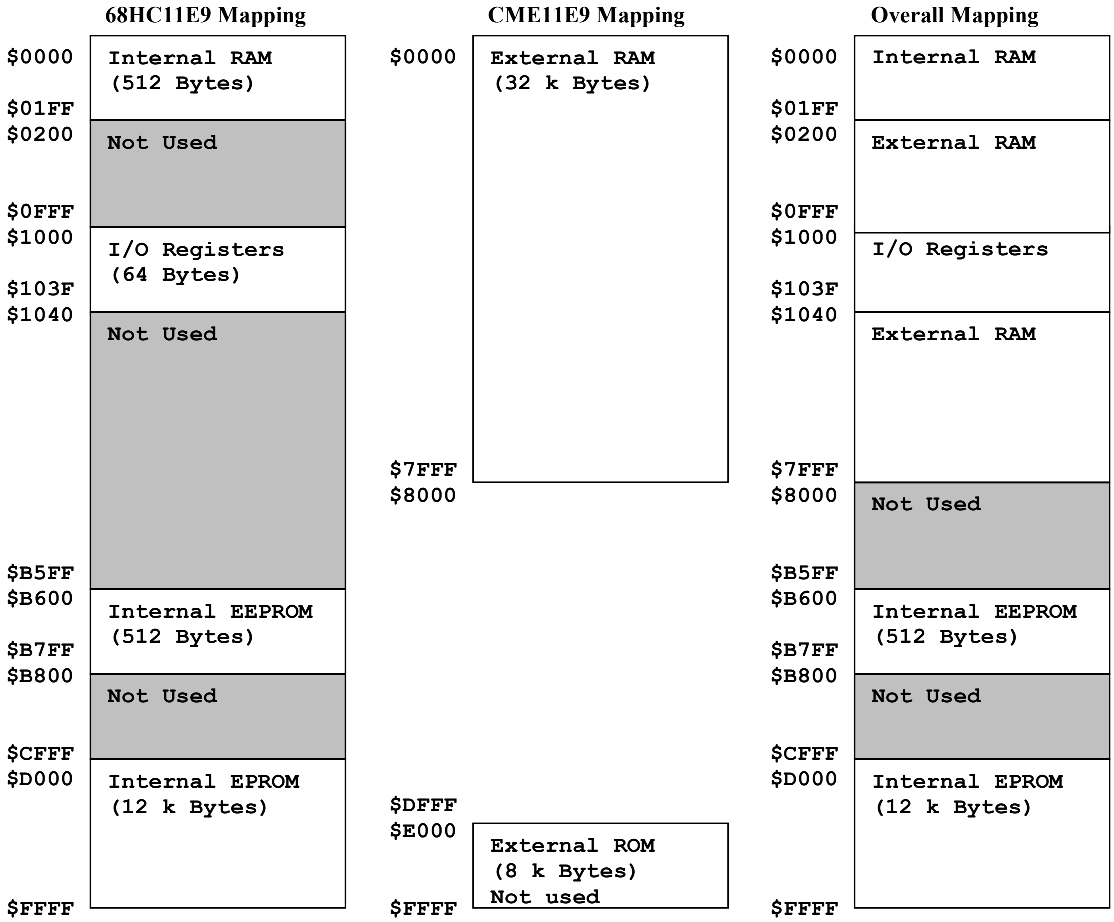


Figure 2: CME11E9-EVBU Memory Map

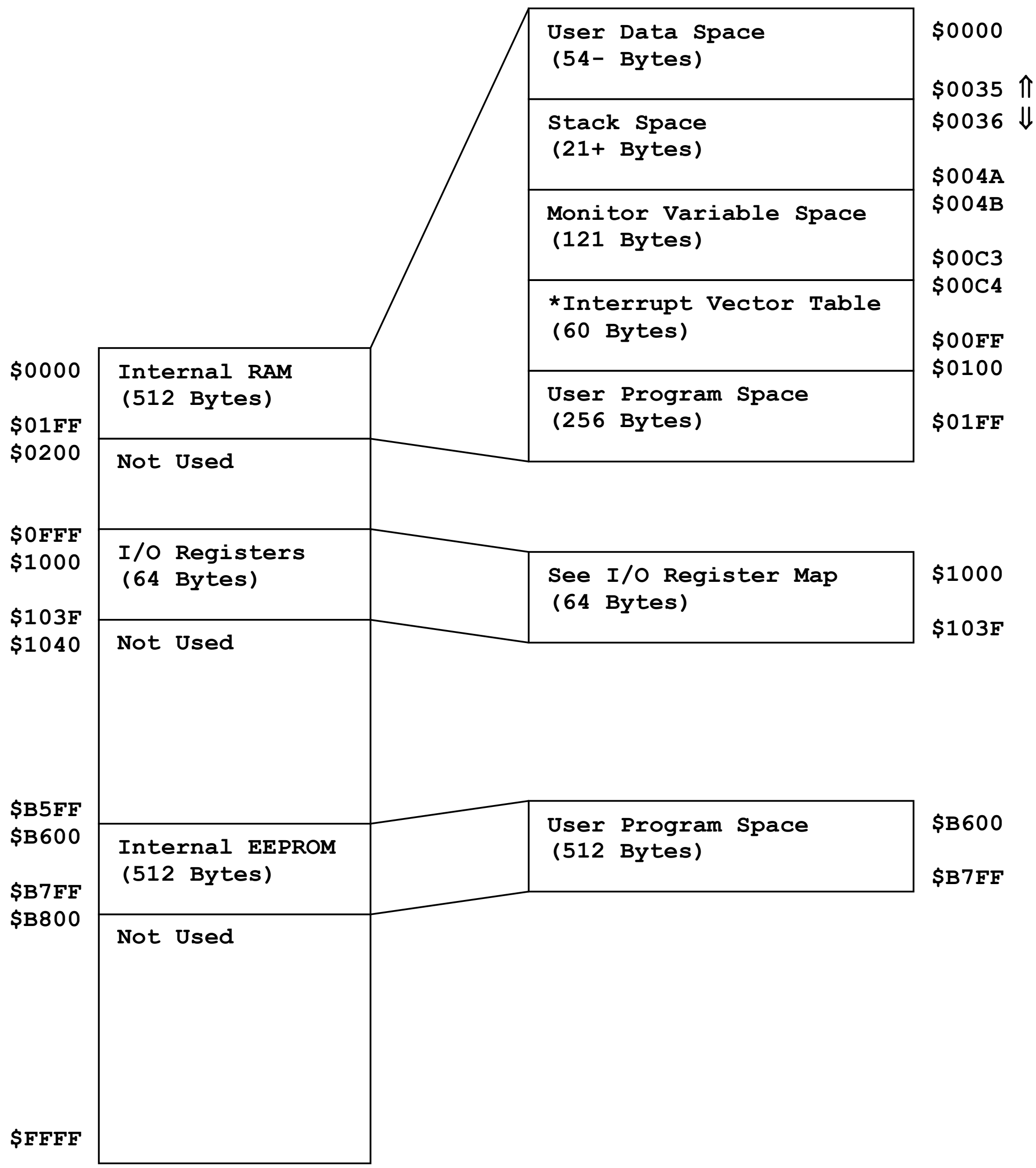


Figure 3: 68HC11E1 Memory Map

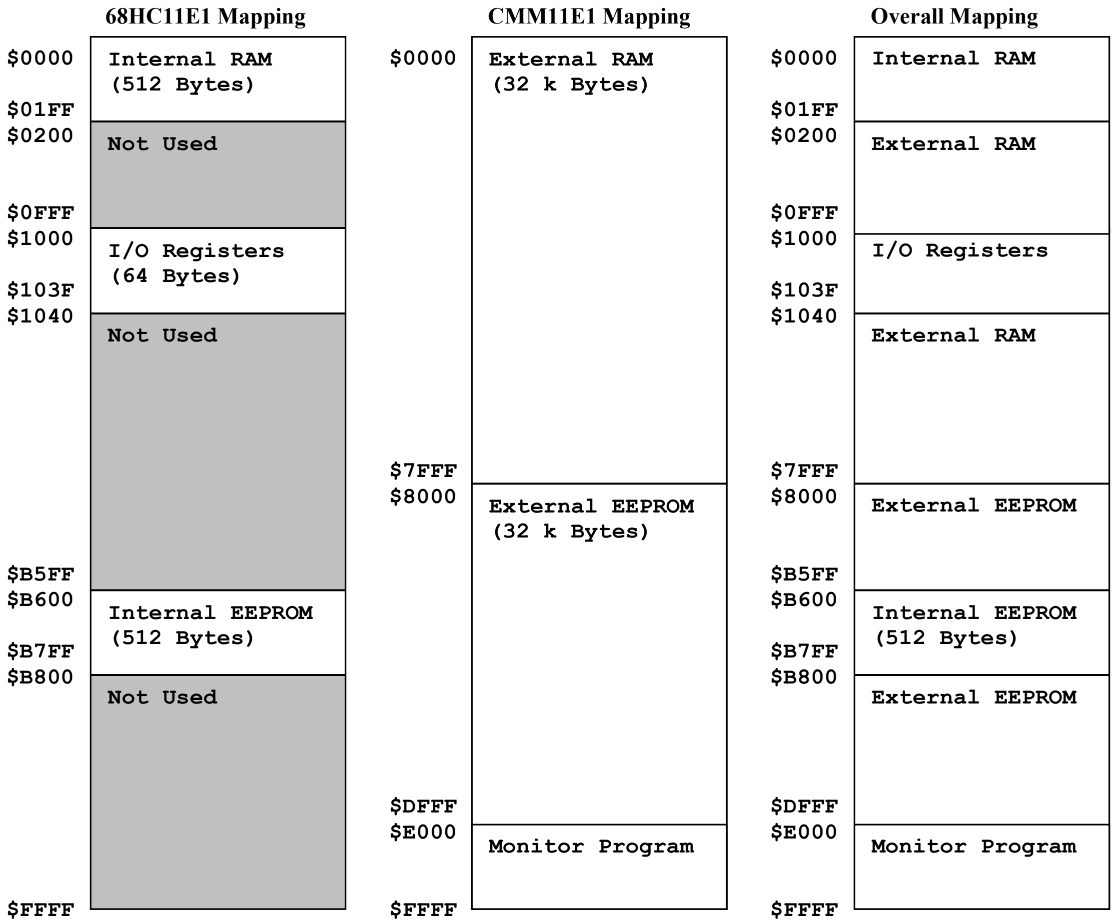


Figure 4: CMM11E1-EVBU Memory Map

2 Communication between PC and EVBU

2.1 Overview

- Monitor program
  - Stored in permanent memory
  - Provides easy access to memory/hardware on the microcontroller
  - The monitor program on the EVBU is called BUFFALO
  - A special Wallace monitor program was written for the Wallace robots.
    - The Wallace monitor program allows easy access to the I/O on the robot
    - BUFFALO provides more general access to the I/O on the 68HC11
- Standard Communication
  - On the PC, the standard input device is the keyboard and the standard output device is the monitor.
  - The serial port on the EVBU is used as the standard input and output device
  - The monitor programs use the standard I/O for communicating
  - The PC must be connected to the EVBU via a serial cable
  - A terminal emulation program (such as HyperTerminal) can be used on the PC to transmit and receive data across the serial port
  - The communication protocols on the PC and EVBU must match
    - 9600 baud, 8 data bits, no parity, 1 stop bit
- Loading a program
  - The executable code must be copied from the host computer (PC) to the memory on the target computer (microcontroller/EVBU)
  - The S19 file format is not true executable code
    - The S19 file is in a special format for transmitting to BUFFALO
    - BUFFALO translates the S19 format into executable code and stores it in the appropriate memory locations
  - To load an S19 file:
    - Use the **LOAD T** command to prepare BUFFALO to receive an S19 file
    - Transmit the S19 as a text file
- Executing a program
  - The 68HC11 is a serial processor (it can only execute one program at a time)
  - To execute a program
    - Use the **GO 1040** command to transfer program execution from BUFFALO to the beginning of your program.
    - When your program ends, program execution will automatically to transferred back to BUFFALO

2.2 Block Diagram

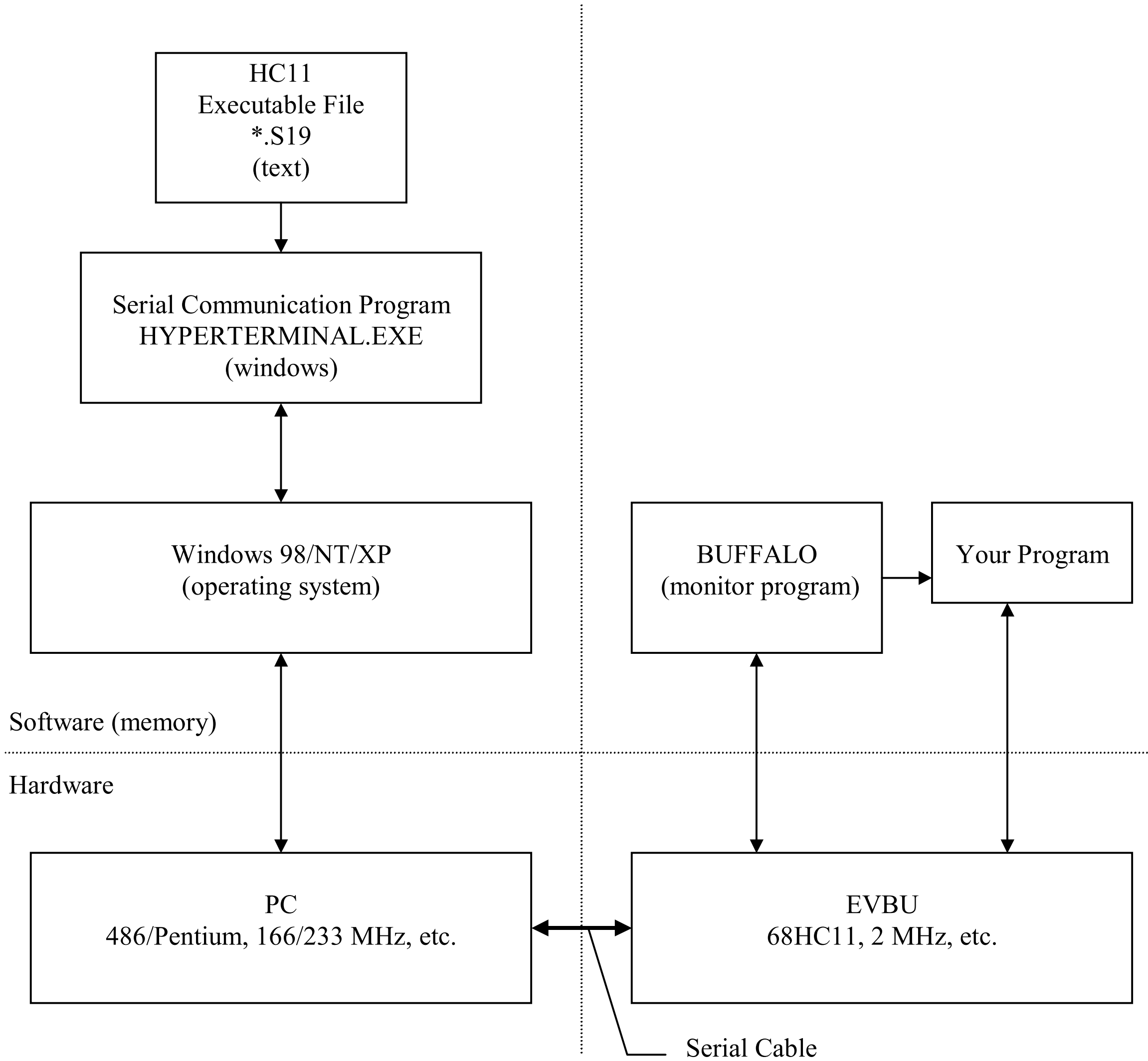


Figure 5: Block Diagram for Communication between PC and EVBU

3 Gcc 68HC11 Compiler

3.1 Overview

- Gcc 68HC11 compiler used to compile C code for the 68HC11 microcontroller.
- It is a 16 bit compiler (an integer/word is 16 bits)
- The 68HC11 is not a floating-point processor. The compiler can handle floating-point math but the code becomes large and time consuming. Do not use floating-point variables or math.
- Use the GCC 68HC11 DOS Prompt and **gcc6811** batch file to compile programs.
- Compiler creates an S19 file.

gcc6811 test.c

- Compiler and linker options pre-configured in **gcc6811** batch file.
- Default starting address for programs is 1040.

3.2 Library Functions

- You cannot use most of the C library functions you are familiar with. The functions are either not available, require too much memory, and/or consume too much time.
- Unable to use standard I/O functions (`printf`, `scanf`, etc.).
- All library functions that you use become part of your program and must be downloaded into memory on the EVBU.

3.3 BUFFALO Library Functions

- BUFFALO program organized into subroutines (modular programming)
- BUFFALO already stored in ROM on the EVBU
- You can access BUFFALO subroutines if you know the address and parameter format
- Use Buffalo library functions to take advantage of BUFFALO subroutines
- Refer to BUFFALO Library Documentation (include `buffalo.h` header file)

3.4 Sample Source Code

```
standard.c (DOS)
buffalo.c (CMM11E1, CME11E9, simulator)
int.c (CMM11E1, CME11E9, simulator)
float.c (CMM11E1, CME11E9, simulator)
```

4 I/O Registers

4.1 Overview

- The I/O registers are used to control the I/O features on the 68HC11.
- Each register has a specific purpose.
- The bits in some registers also have a specific purpose.
- Some registers can be modified by the user while others cannot.
- The registers are accessed via memory mapping.

4.2 Memory Mapped I/O Register

- The I/O registers on the 68HC11 are mapped to specific memory locations.
- Read memory location to determine register value.
- Write to memory location to change register value (if it can be changed).

Table 1: I/O Registers Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1000	PORTA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
1001	reserved								
1002	PIOC								
1003	PORTC	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
1004	PORTB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
1005	PORTCL								
1006	reserved								
1007	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
1008	PORTD	0	0	PD5	PD4	PD3	PD2	PD1	PD0
1009	DDRD	0	0	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
100A	PORTE	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
100B	CFORC								
100C	OC1M	OC1M7	OC1M6	OC1M5	OC1M4	OC1M3	0	0	0
100D	OC1D	OC1D7	OC1D6	OC1D5	OC1D4	OC1D3	0	0	0
100E	TCNT(Hi)	15	14	13	12	11	10	9	8
100F	TCNT(Lo)	7	6	5	4	3	2	1	0
1010	TIC1(Hi)	15	14	13	12	11	10	9	8
1011	TIC1(Lo)	7	6	5	4	3	2	1	0
1012	TIC2(Hi)	15	14	13	12	11	10	9	8
1013	TIC2(Lo)	7	6	5	4	3	2	1	0
1014	TIC3(Hi)	15	14	13	12	11	10	9	8
1015	TIC3(Lo)	7	6	5	4	3	2	1	0
1016	TOC1(Hi)	15	14	13	12	11	10	9	8
1017	TOC1(Lo)	7	6	5	4	3	2	1	0
1018	TOC2(Hi)	15	14	13	12	11	10	9	8
1019	TOC2(Lo)	7	6	5	4	3	2	1	0
101A	TOC3(Hi)	15	14	13	12	11	10	9	8
101B	TOC3(Lo)	7	6	5	4	3	2	1	0
101C	TOC4(Hi)	15	14	13	12	11	10	9	8
101D	TOC4(Lo)	7	6	5	4	3	2	1	0
101E	TI4O5(Hi)	15	14	13	12	11	10	9	8
101F	TI4O5(Lo)	7	6	5	4	3	2	1	0
1020	TCTL1	OM2	OL2	OM3	OL3	OM4	OL4	OM5	OL5
1021	TCTL2	EDG4B	EDG4A	EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A
1022	TMSK1	OC1I	OC2I	OC3I	OC4I	I4O5I	IC1I	IC2I	IC3I
1023	TFLG1	OC1F	OC2F	OC3F	OC4F	I4O5F	IC1F	IC2F	IC3F
1024	TMSK2	TOI	RTII	PAOVI	PAII	0	0	PR1	PR0
1025	TFLG2	TOF	RTIF	PAOVF	PAIF	0	0	0	0
1026	PACTL	DDRA7	PAEN	PAMOD	PEDGE	DDRA3	I4O5	RTR1	RTR0
1027	PACNT	7	6	5	4	3	2	1	0
1028	SPCR								
1029	SPSR								
102A	SPDR								
102B	BAUD	TCLR	0	SCP1	SCP0	RCKB	SCR2	SCR1	SCR0
102C	SCCR1	R8	T8	0	M	WAKE	0	0	0
102D	SCCR2	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
102E	SCSR	TDRE	TC	RDRF	IDLE	OR	NF	FE	0
102F	SCDR	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0
1030	ADCTL	CCF	0	SCAN	MULT	CD	CC	CB	CA
1031	ADR1	7	6	5	4	3	2	1	0
1032	ADR2	7	6	5	4	3	2	1	0
1033	ADR3	7	6	5	4	3	2	1	0
1034	ADR4	7	6	5	4	3	2	1	0
1035	BPROT								
1036	reserved								
1037	reserved								
1038	reserved								
1039	OPTION	ADPU	CSEL	IRQE	DLY	CME	0	CR1	CR0
103A	COPRST								
103B	PPROG								
103C	HPRIO								
103D	INIT								
103E	TEST1								
103F	CONFIG								

5 Parallel I/O

5.1 Overview

- Access digital inputs and outputs via corresponding PORT registers.
- Each signal/pin has corresponding bit in PORT register.
- Determine current input and output signal values by reading PORT registers.
- Assign output signal values by writing to PORT registers.
- You cannot change the value of an input signal.

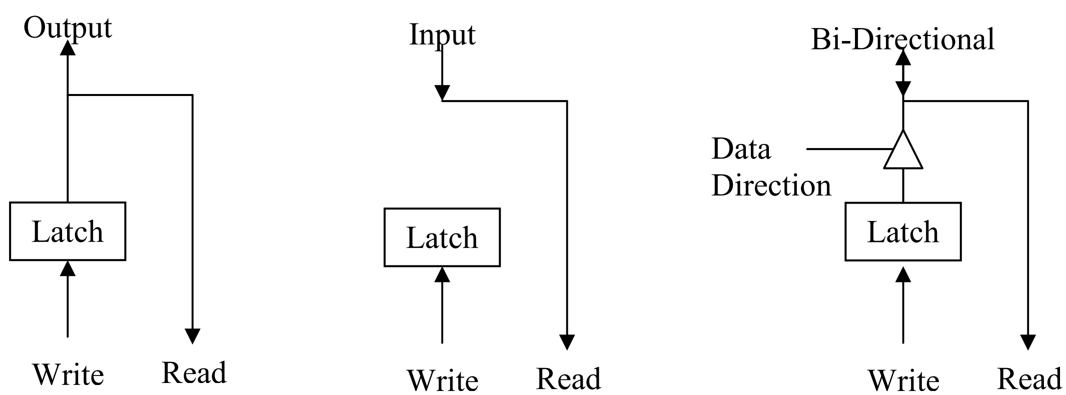


Figure 6: Block Diagram for Parallel I/O Latches

5.2 Output Ports

- Write to I/O register to assign output values.
- Read from I/O register to determine output values.

Port B Summary (single chip mode)

Address	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1004	PORTB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB 0
		↓	↓	↓	↓	↓	↓	↓	↓

5.3 Input Ports

- Read from I/O register to determine input values.
- Writing to I/O register has no effect.

Port E Summary

Address	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
100A	PORTE	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
		↑	↑	↑	↑	↑	↑	↑	↑

5.4 Bi-directional Ports

- Configure direction using data direction (DD) bits. Reset (default) value is 0 (input).
  - 0 = input
  - 1 = output
- Data direction bits grouped into data direction registers (DDR). The DD bits correspond with the bits/pins they control.
- Read from I/O register to determine pin values.
- Write to I/O register to set output pins (input pins are not effected).

Port C Summary (single chip mode)

Address	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1007	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
1003	PORTC	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
		↕	↕	↕	↕	↕	↕	↕	↕

Port D Summary

Address	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1009	DDRD	0	0	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
1008	PORTD	0	0	PD5	PD4	PD3	PD2	PD1	PD0
				↕	↕	↕	↕	↕	↕

Port A Summary

Address	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1026	PACTL	DDRA7	-	-	-	DDRA3	I4/O5	-	-
1000	PORTA	PA7	PA6	PA5	PA4	PA3*	PA2	PA1	PA0
		↕	↓	↓	↓	↕	↑	↑	↑

*PA3 has two control pins. If the I4/O5 bit is equal to 0 (default) the PA3 pin is configured as an output. The OM5 and OL5 bits control the output state. If OC5 is disabled (OM5 = 0 and OL5 = 0) or if PA3 is configured for input capture (I4/O5 = 1) then the DDA3 bit determines the direction of the PA3 pin.

6 I/O Register C Definitions

6.1 Overview

- The `hc11e9.h` header file contains the definitions for the 68HC11 I/O registers.

```
#include <hc11e9.h>
```

6.2 Register Defines

- I/O registers are defined as a de-referenced pointer to their corresponding address.
- You do not need to place this in your source code (defined in `hc11e9.h` header file)

```
#define PORTA   *(volatile unsigned char *)0x1000
#define PORTB   *(unsigned char *)0x1004
```

6.3 Register Usage

- Use I/O registers in your program like variables.
- You can use them in assignments, conditional statements, etc.
- You can always change the value of a variable. They do not change unless you change them.
- You cannot always change the value of an I/O register. Some of them can change on their own.

```
int     x, y;

x = PORTB;
y = PORTB;
PORTB = 0xE5;
if (PORTB == 0x40)
...
```

Variables x and y will always be the same Bits (pins) PB7, PB6, PB5, PB2, PB0 set to a 1 (high) Bits (pins) PB4, PB3, PB1 set to a zero (low) Checks if PB6 = 1 and all other pins = 0

```
x = PORTE;
y = PORTE;
PORTE = 0xE5;
if (PORTE == 0x40)
...
```

Variables x and y may be different Has no effect on bits (pins) Checks if PE6 = 1 and all other pins = 0

6.4 Bit Defines

- I/O register bits are defined as the corresponding bit mask.
- I/O register bit defines are not associated with specific registers (all bit 7 bits defined the same). You must ensure correct bit/register combination usage.
- Use appropriate bit defines instead of actual mask to make code easier to understand.
- You do not need to place this in your source code (defined in `hc11e9.h` header file)

```
#define bit7    0x80
#define bit6    0x40

#define PB7     bit7
#define PB6     bit6

#define PE7     bit7
#define PE6     bit6
```

6.5 Bit Usage

- You cannot access bits directly. You must go through the I/O register.
- The bit defines are not variables. They do not represent the value of the bit.
- The actual bit values are in the I/O register. Use the bit defines and bit wise operations to access specific bits in the I/O register.

```
/* Invalid C statements */
PB7 = 1;                        /* Same as 0x80 = 1 */
PB7 = 0;                        /* Same as 0x80 = 0 */

/* Modify entire register */
PORTB = 0x04;                   /* Cryptic */
PORTB = PB2;                    /* More understandable */
PORTB = (PB3 | PB4);            /* Combine masks */

/* Set specific bits to 1, leave other bits alone */
PORTB = PORTB | PB3;            /* PB3 = 1 */
PORTB |= PB3;                   /* Shortcut notation */
PORTB |= (PB3 | PB4);           /* PB3 = 1 , PB4 = 1 */

/* Set specific bits to 0, leave other bits alone */
PORTB = PORTB & ~PB3;           /* PB3 = 0 */
PORTB &= ~PB3;                  /* Shortcut notation */
PORTB &= ~(PB3 | PB4);          /* PB3 = 0 , PB4 = 0 */

/* Check specific bits */
if ( (PORTB & PB3) == 0 )       ...     /* PB3 = 0 ? */
if ( (PORTB & PB3) == PB3 )     ...     /* PB3 = 1 ? */
```

IMPORTANT NOTE:

- When referring to bits (pins) outside the context of C code, it is valid to say PB7 = 1.
- When referring to bits in the context of C code, it is **invalid** to say `PB7=1`.

7 Clocks and Counters

7.1 Overview

- A clock is a periodic signal used to synchronize events.
  - The CSU requires a clock to synchronize the execution of instructions.
  - Clocks are often used in conjunction with hardware counters.
- A hardware counter increments (counts) on the rising edge of the clock.
  - Counters have a limited range based on the number of bits in the counter.
  - When the counter goes from all ones to all zeros it is said to overflow.
  - Specific counters are connected to different clocks, resulting in different counting rates.
- The timer functions and other I/O features require clocks and counters to perform their operations.

7.2 Clocks

- EXTAL
  - Input to 68HC11. Supplied by external circuitry.
- XTAL
  - Output from the 68HC11 for use by external devices.
  - Highest internal clock frequency. Use for synchronization of instructions.
  - The stop disable bit is used to put the 68HC11 to sleep (turn off XTAL).
- E Clock
  - Internal 68HC11 clock.
  - Down-sampled XTAL clock by four (1/4 frequency).
  - Base clock for timer functions. A machine cycle, or tick, is equal to one period of the E clock.
  - Other clocks on 68HC11 based on E-clock.
- EVBU Board
  - External 8 MHz crystal oscillator
    - EXTAL = 8 MHz
    - XTAL = 8 MHz
    - E Clock = 2 MHz (1 tick = 0.5 μsec)

7.3 Free Running Clock and Counter

7.3.1 Overview

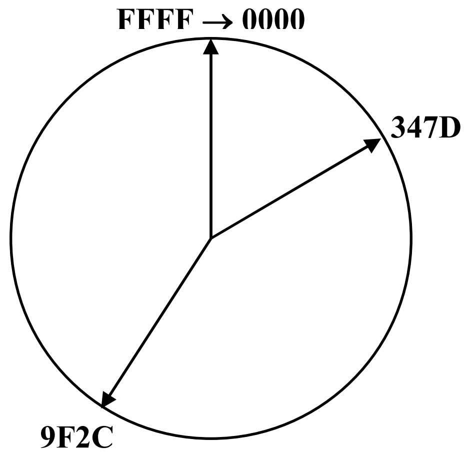
- The free running clock is based off of the E clock.
- Configuration bits PR1 and PR0 are timed write-once bits (you can not change them).
- The free running counter (TCNT) is a 16 bit I/O register.
- The timer functions are based on the free running clock/counter.

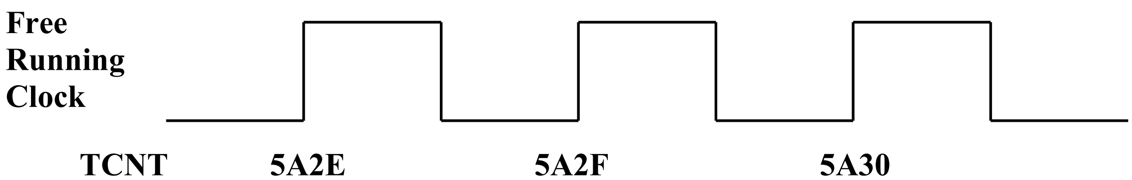
Free Running Clock and Counter (TCNT)

PR1	PR0	↓Scale	Frequency (kHz)	Period (μs)	Overflow (ms)
0	0	1	2000	0.5	32.768
0	1	4	500	2.0	131.072
1	0	8	250	4.0	262.144
1	1	16	125	8.0	524.288

Free Running Counter I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TCNT(Hi)	15	14	13	12	11	10	9	8
TCNT(Lo)	7	6	5	4	3	2	1	0
TMSK2	-	-	-	-	-	-	PR1	PR0





7.3.2 Sample Source Code

```
tcnt.c (CMM11E1, CME11E9, simulator)
tcntdelay.c (CMM11E1, CME11E9)
```

7.4 Real-Time Interrupt Clock

- The real-time clock is based off of the E clock.
- Configuration bits RTR1 and RTR0 can be modified.
- The period of the real-time clock is on the order of milliseconds, which corresponds to typical update rates for real-time applications.

Real-Time Interrupt (RTI) Clock

RTR1	RTR0	↓Scale	Period (ms)
0	0	2^{13}	4.096
0	1	2^{14}	8.192
1	0	2^{15}	16.384
1	1	2^{16}	32.768

Real-Time Clock I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PACTL	-	-	-	-	-	-	RTR1	RTR0

7.5 Software Timing Measurements

- The free running counter can be used to perform timing measurements.
- The accuracy of the timing measurements is limited by the period of the free running clock and the size of the free running counter.
    - Smallest measurable time interval is 1 clock period (0.5 μsec).
    - Largest measurable time interval is 1 counter cycle (32.768 msec).
    - Only relative time can be measured (i.e. second hand only).
- In addition, there are limitations and considerations that need to be taken into account depending on the method of measurement.
    - Software always requires time to execute.
    - All assembly instructions take at least 1 tick (0.5 μsec) to execute. Most assembly instructions take several ticks or more.
    - C statements typically correspond to several assembly instructions.
    - Hardware is capable of performing operations faster than software.

7.5.1 Example – Measuring a Pulse Width

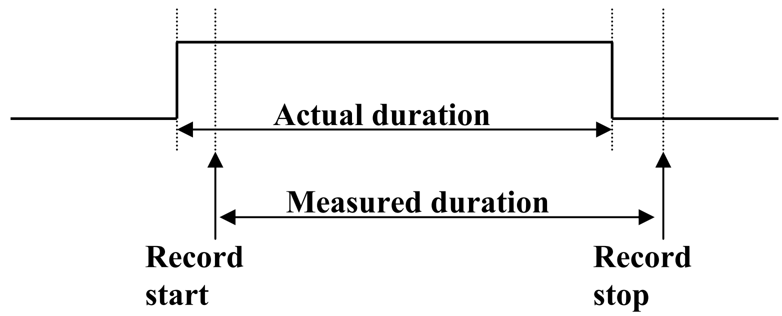
- Objective
  - Measure the pulse width of the signal shown below

- Strategy
  - Connect signal to an input pin (pick PA2).
  - Wait for rising edge. Record time.
  - Wait for falling edge. Record time.
  - Take the difference to calculate pulse width.

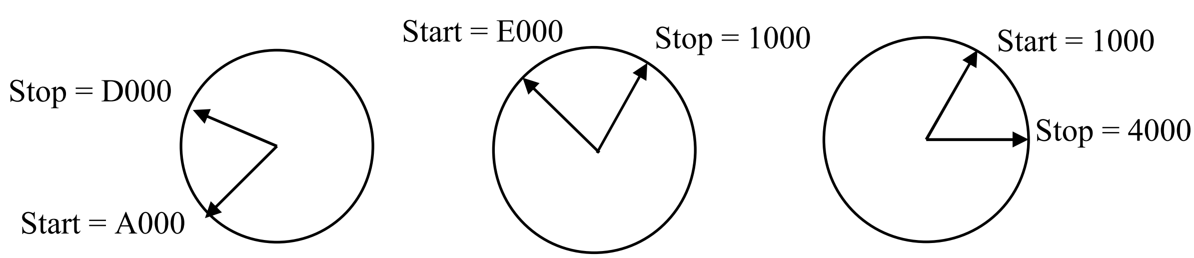
- Sample Source Code

```
tcntmpw.c (CME11E9, simulator)

unsigned start, stop, width;    /* Stop watch variables */

while ( (PORTA & PA2) == PA2);  /* Ensure start during low part */

while ( (PORTA & PA2) == 0);    /* Wait until PA2 high */
start = TCNT;                   /* Record start time */
while ( (PORTA & PA2) == PA2);  /* Wait until PA2 low */
stop = TCNT;                    /* Record stop time */
width = stop - start;           /* Calculate pulse width(ticks) */
```

- Observations
  - Time is always positive (unsigned), relative, and in ticks.
  - Delay between detection of edge and recording of time.



  - Unsigned math can handle one counter overflow.
  - Does not detect wrap around times.



  - Actual pulse width could be 3000, 13000, 23000, …

- Questions?
  - What is the minimum pulse width that can be measured using this method?
  - What is the maximum pulse width that can be measured using this method?

7.5.2 Example – Generating a Pulse Width

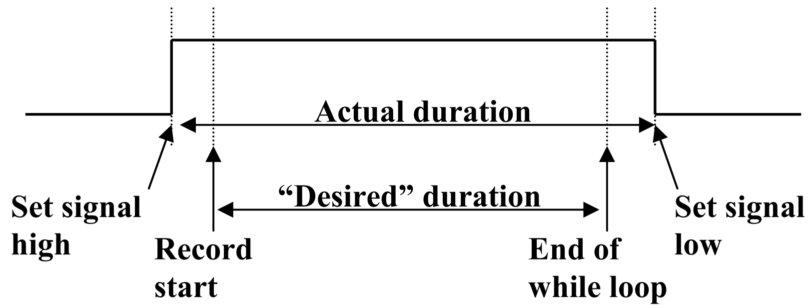
- Objective
  - Generate the signal shown below.

- Strategy
  - Connect signal to an output pin (pick PA6).
  - Set signal high.
  - Record time.
  - Wait for pulse width duration to elapse.
  - Set signal low.

- Sample Source Code

```
tcntgpw.c (CME11E9, simulator)

unsigned start, width;          /* Stop watch variables */

width = 100;                    /* 50 usec pulse width */

PORTA |= PA6;                   /* Set PA6 high */
start = TCNT;                   /* Record start time */
while ( (TCNT-start) < width);  /* Wait */
PORTA &= ~PA6;                  /* Set PA6 low */
```

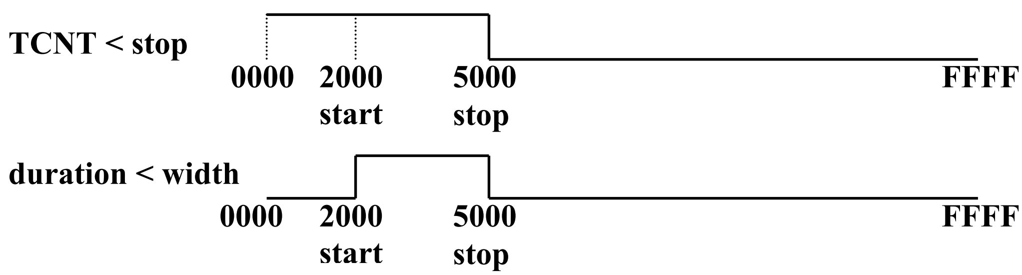
- Observations
  - Delay between setting signal high and recording of start time.
  - Delay between end of while loop and setting signal low.



  - Cannot use equality comparisons.
  - Must use proper conditional statement in order to handle counter overflow.

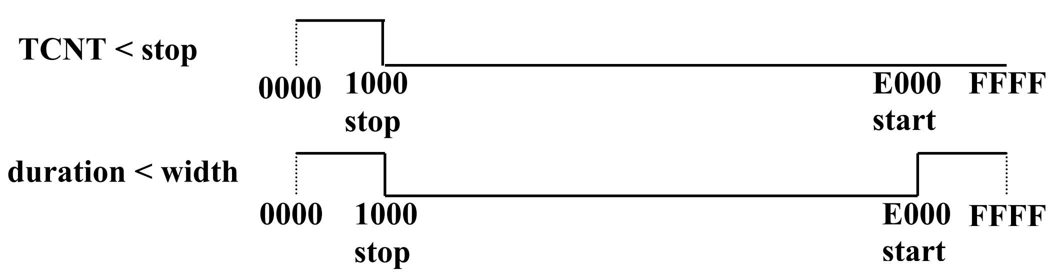
start = 2000, width = 3000, stop = start + width = 5000

TCNT	duration = TCNT-start	TCNT < stop	duration < width
4000	2000	True	True
6000	4000	False	False
1000	F000	True	False



start = E000, width = 3000, stop = start + width = 1000

TCNT	duration = TCNT-start	TCNT < stop	duration < width
F000	1000	False	True
0000	2000	True	True
2000	4000	False	False



- Questions?
  - What is the minimum pulse duration that can be generated using this method?
  - What is the maximum pulse duration that can be generated using this method?

8 Events and Flags

8.1 Overview

- How does software and hardware "communicate" with each other?
  - Software "communicates" with hardware by modifying I/O registers.
  - Hardware "communicates" with software by modifying I/O registers.
    - Software must check I/O registers to "notice" information.
    - Hardware sets a flag bit to indicate an event occurred.
- An event occurs whenever specific conditions are met.
  - Every event has its own set of conditions.
    - Alarm clock event occurs whenever the actual time matches the preset alarm time.
    - Light turns on when switch is thrown.
  - Event conditions are checked by hardware.
    - Conditions can be checked "instantaneously".
    - It does not take up processor time.
- A flag is a bit located in an I/O register that indicates whether or not an event has occurred.
  - The purpose of a flag bit is to inform "software" that an event has occurred.
  - It is the responsibility of the "software" to check the flag bit.
  - Every event has its own flag bit.
  - A flag can only be set to a 1 by hardware. Software cannot set a flag.
  - Most flags can be cleared (set to a 0) by software by writing a 1 to the flag bit.
  - Flags are cleared on RESET.
  - Flag bits are denoted by <event name>F (i.e. the RTI flag is RTIF).
- There are two methods for responding to events: polling technique and interrupt technique.
  - Polling technique – program checks flags to determine if event occurs
  - Interrupt technique – hardware generates an interrupt whenever event occurs

8.2 Polling Technique

8.2.1 Overview

- Software periodically checks to see if a flag is set.
    - Good – Program controls when the flag is checked.
    - Bad – Can be a significant delay between event and when software checks the flag.

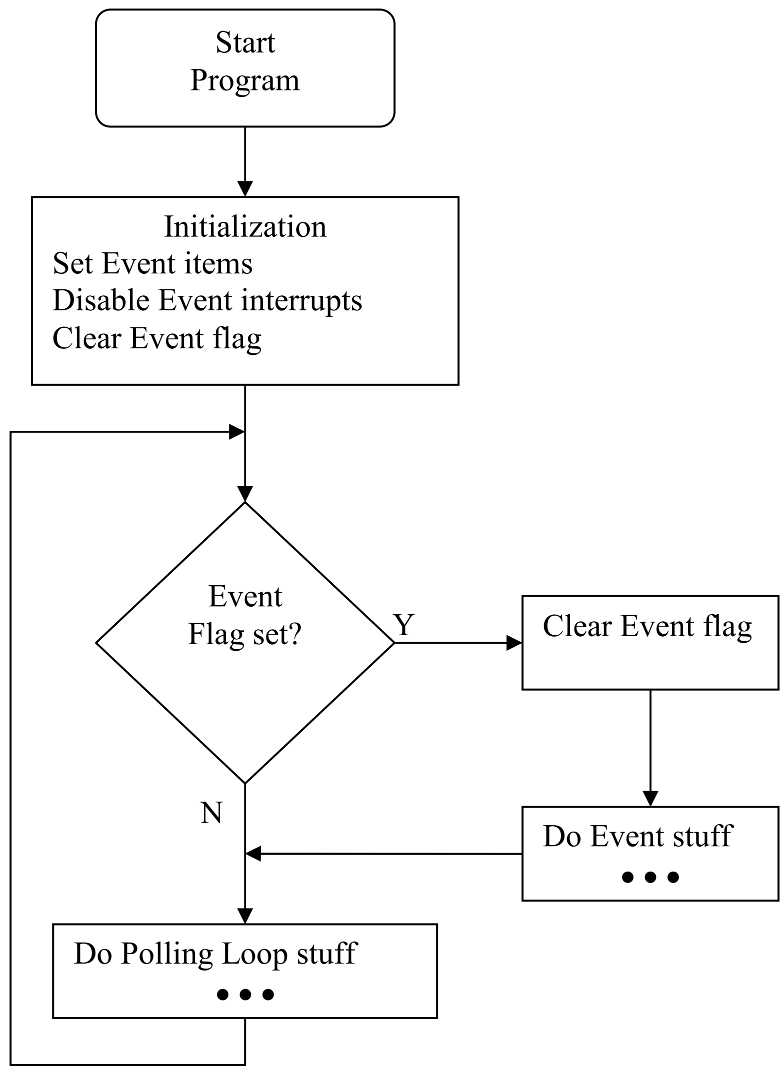


Figure 7: Flow Chart for Polling Technique

8.2.2 Sample Source Code

```
tofdelay.c (CMM11E1, CME11E9)
tof.c (CMM11E1, CME11E9, simulator)
```

8.3 Interrupt Technique

8.3.1 Overview

- Hardware will interrupt the execution of the program when an event occurs.
- An interrupt service routine (ISR) associated with the event will automatically be executed.
    - ISRs differ from normal subroutines in how they are called and how they return.
    - An ISR cannot be called directly in the program.
- When the ISR is completed, the program will resume from where it was interrupted.
    - Good – No need to waste time checking flags.
    - Good – More robust/structured programming.
    - Not so Bad – There is still a slight delay in software response.
    - Bad – Too many interrupts cause the program to starve.

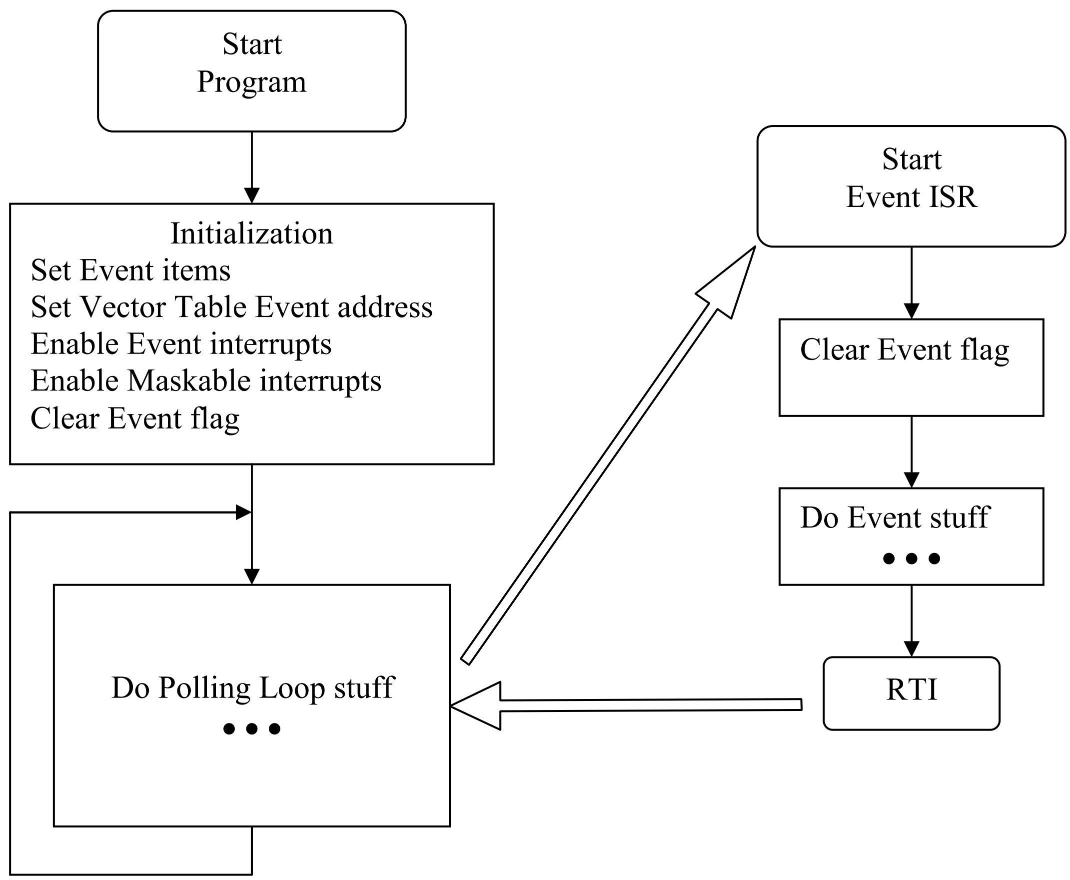


Figure 8: Flow Chart for Interrupt Technique

8.3.2 Sample Source Code

`toi.c (CMM11E1, CME11E9, simulator)`

8.3.3 Terminology

- Synchronous / Asynchronous
    - Synchronous interrupts – current instruction is completed prior to executing ISR. This is a recoverable interrupt.
    - Asynchronous interrupts – current instruction is aborted and ISR is executed immediately. This is a non-recoverable interrupt.
- Maskable / Non-Maskable interrupts
    - Maskable interrupts – the interrupt can be disabled (masked).
    - Non-Maskable interrupts – the interrupt cannot be disabled (masked).
        - The I bit in the CCR register is used to enable (0)/disable (1) “all” maskable interrupt
        - The X bit in the CCR register is used to enable (0)/disable (1) the XIRQ interrupt
            - Once enabled the XIRQ cannot be disable except by a RESET.
        - There are additional individual enable bits for each maskable interrupt (and some “non-maskable” interrupts.
- Priority
    - Determines the order in which multiple events/interrupts that occur at the same time will be handled.
- Vectors
    - A vector is an address that corresponds to a subroutine (points to a subroutine)
    - An interrupt vector (IV) contains the address of an ISR.
    - An interrupt vector table (IVT) contains a list of interrupt vectors. Each vector in the table corresponds to a specific interrupt event. The IVT is located in ROM.
    - A pseudo-interrupt vector (PIV) contains a jump statement to an ISR.
    - A pseudo-interrupt vector table (PIVT) contains a list of pseudo-interrupt vectors. The PIVT is located in RAM to allow assignment of ISRs during the development process.
    - The IVT is mapped to the PIVT on the EVBU.

```
                Pseudo-Vector Table
$00C4
  ...
$00F4   7E  ┐
$00F5   ??  ├   Start of SWI ISR
$00F6   ??  ┘   (JMP xxxx)
  ...
$00FF

                Vector Table
$FFC0
  ...
$FFF6   00  ┐
$FFF7   F4  ┘   Extended address of SWI ISR (00F4)
  ...
$FFFF
```

Figure 9: Interrupt Vector Example

Table 2: Interrupt Vector Summary

Vector Address (2 bytes)	Pseudo-Vector Address (3 bytes)	Source in order of priority (low to high)	CCR Mask Bit	Local Mask Bit	Synchronous
FFD6	00C4	Serial Communication Interface (SCI)	I	RIE TIE TCIE ILIE	Y
FFD8	00C7	Serial Peripheral Interface (SPI)	I	SPIE	Y
FFDA	00CA	Pulse Accumulator Input Edge	I	PAII	Y
FFDC	00CD	Pulse Accumulator Overflow	I	PAOVI	Y
FFDE	00D0	Timer Overflow	I	TOI	Y
FFE0	00D3	Timer Input Capture 4 / Timer Output Compare 5	I	I4/OC5I	Y
FFE2	00D6	Timer Output Compare 4	I	OC4I	Y
FFE4	00D9	Timer Output Compare 3	I	OC3I	Y
FFE6	00DC	Timer Output Compare 2	I	OC2I	Y
FFE8	00DF	Timer Output Compare 1	I	OC1I	Y
FFEA	00E2	Timer Input Capture 3	I	IC3I	Y
FFEC	00E5	Timer Input Capture 2	I	IC2I	Y
FFEE	00E8	Timer Input Capture 1	I	IC1I	Y
FFF0	00EB	Real-Time Interrupt	I	RTII	Y
FFF2	00EE	IRQ	I		Y
FFF4	00F1	XIRQ	X		Y
FFF6	00F4	Software Interrupt (SWI)			Y
FFF8	00F7	Illegal Opcode			
FFFA	00FA	Computer Operating Properly (COP)		NOCOP	
FFFC	00FD	Clock Monitor		CME	
FFFE		RESET (Bootloader Start)			

8.3.4 Sequence of Operation

- When a synchronous interrupt occurs the HW sets an interrupt flag.
- When the current instruction cycle is completed, the HW processes the interrupt.
    - If multiple interrupts occurred, the interrupt with the highest priority is processed first. The other interrupts will remain in a queue to be processed later.
- The CPU registers are pushed onto the stack to be preserved.
- The I and X bits in the CCR register are set to prevent the ISR from being interrupted.
    - If interrupts occurs during an ISR they will be placed in the queue.
- The corresponding IV is fetched from the IVT and placed in the PC (ISR starts executing).
- The RTI instruction restores all the CPU registers to their pre-interrupted state.

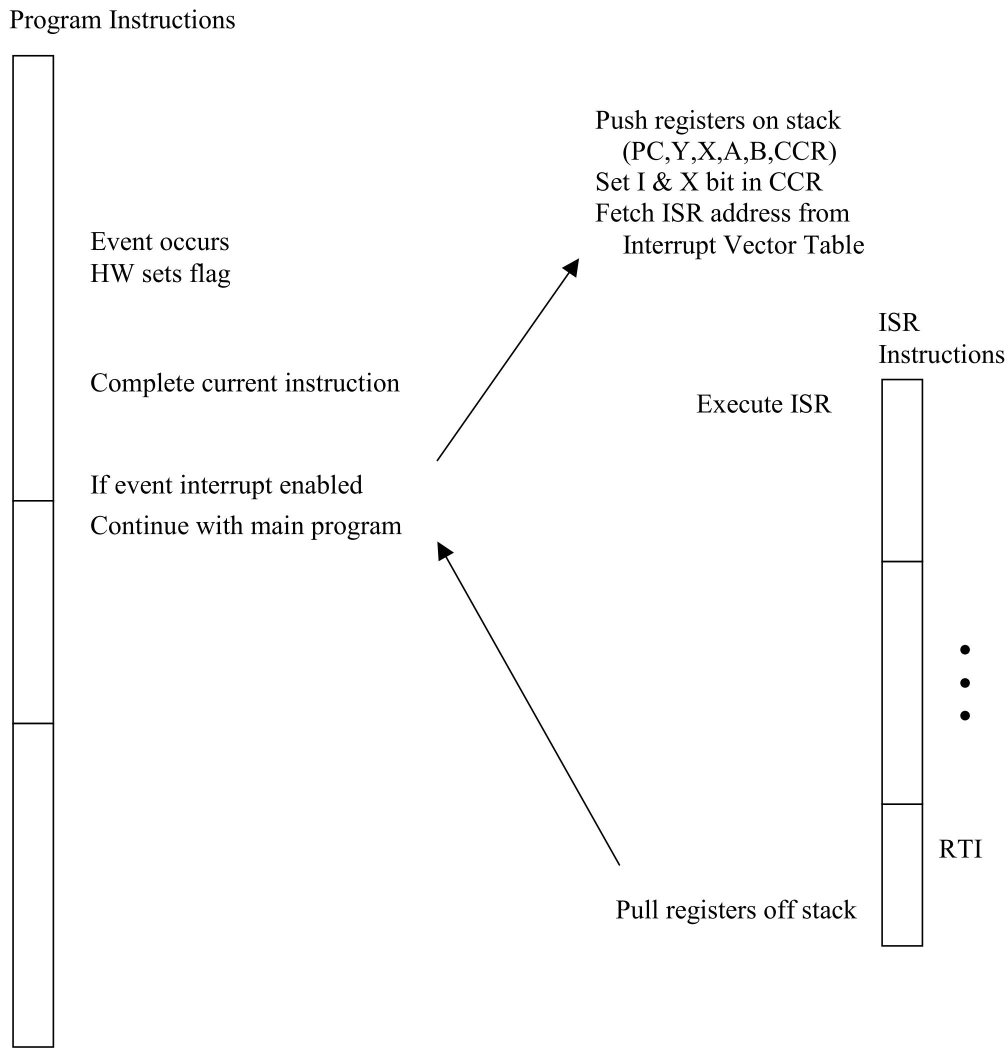
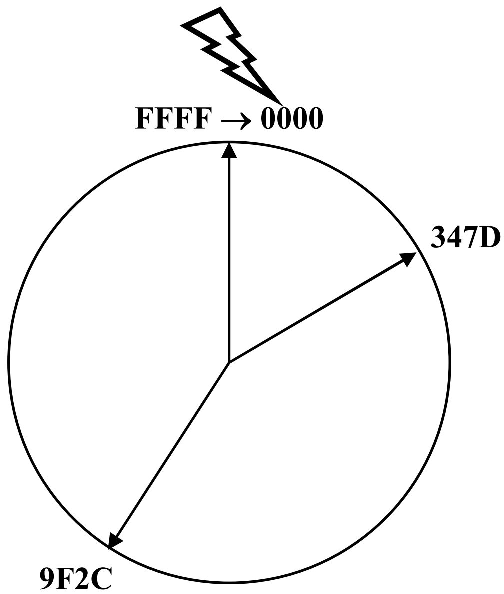


Figure 10: Interrupt Sequence of Operation

9 Timer Overflow

9.1 Overview

- Event occurs when the free running counter overflows.
- Periodic event.
- No external pins associated with event.
- Operates at same time scale as real time interrupt event.



Timer Overflow Event

Flag	Interrupt Enable	Control Registers	Pins
TOF (TFLG2)	TOI (TMSK2)	PR1, PR0 (TMSK2)	None

Timer Overflow I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TMSK2	TOI	-	-	-	-	-	PR1	PR0
TFLG2	TOF	-	-	-	-	-	-	-

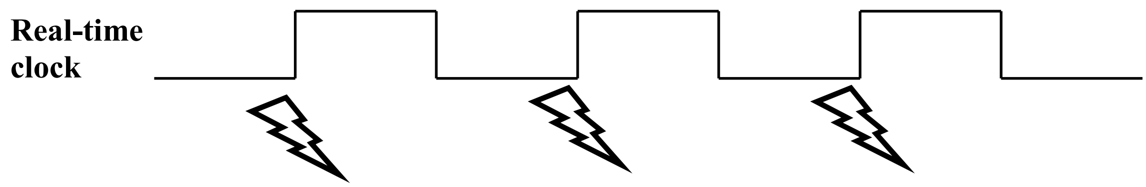
9.2 Sample Source Code

`tocntr.c (CMM11E1, CME11E9, simulator)`

10 Real-Time Interrupt

10.1 Overview

- Event occurs when the real time clock ticks.
- Periodic event.
- No external pins associated with event.



Real-Time Interrupt Event

Flag	Interrupt Enable	Control Registers	Pins
RTIF (TFLG2)	RTII (TMSK2)	RTR1, RTR0 (PACTL)	None

Real-Time Interrupt I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TMSK2	-	RTII	-	-	-	-	-	-
TFLG2	-	RTIF	-	-	-	-	-	-
PACTL	-	-	-	-	-	-	RTR1	RTR0

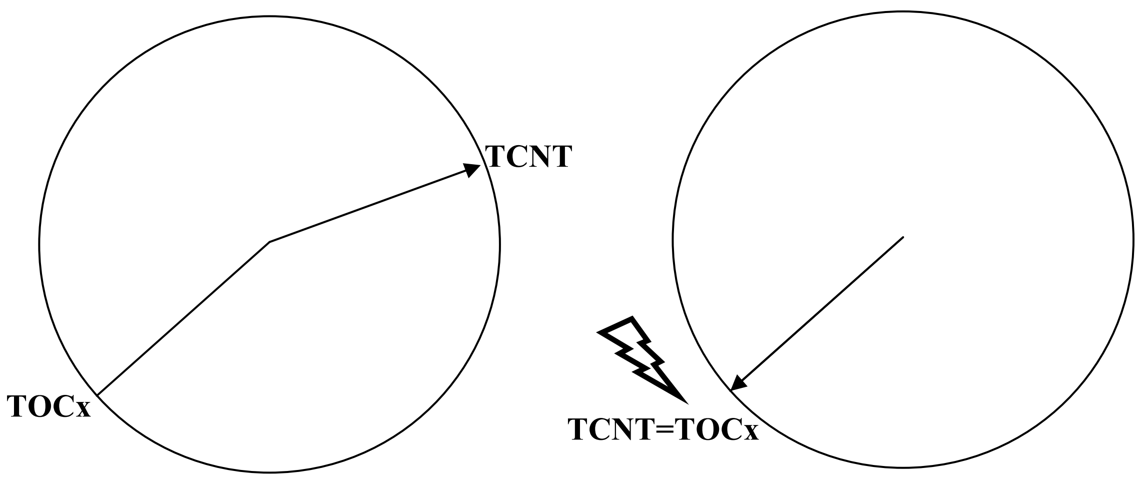
10.2 Sample Source Code

`rticntr.c (CMM11E1, CME11E9, simulator)`

11 Output Compares

11.1 Overview

- Event occurs when TCNT equals TOCx (Output **Compares**).
- Alarm clock type event.
- External pin(s) associated with each event.
- Output pin(s) can be changed automatically when event occurs (**Output** Compares).



Output Compare Events (1…5)

Flag	Interrupt Enable	Control Registers	Pins
OC1F (TFLG1)	OC1I (TMSK1)	(OC1M,OC1D) (TOC1)	OC1…OC5 PA7…PA3
OCxF (TFLG1)	OCxI (TMSK1)	OMx,OLx (TCTL1) (TOCx)	OC2…OC5 PA6…PA3

Output Compare I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OC1M	OC1M7	OC1M6	OC1M5	OC1M4	OC1M3	0	0	0
OC1D	OC1D7	OC1D6	OC1D5	OC1D4	OC1D3	0	0	0
TOC1(Hi)	15	14	13	12	11	10	9	8
TOC1(Lo)	7	6	5	4	3	2	1	0
TOC2(Hi)	15	14	13	12	11	10	9	8
TOC2(Lo)	7	6	5	4	3	2	1	0
TOC3(Hi)	15	14	13	12	11	10	9	8
TOC3(Lo)	7	6	5	4	3	2	1	0
TOC4(Hi)	15	14	13	12	11	10	9	8
TOC4(Lo)	7	6	5	4	3	2	1	0
TI4O5(Hi)	15	14	13	12	11	10	9	8
TI4O5(Lo)	7	6	5	4	3	2	1	0
TCTL1	OM2	OL2	OM3	OL3	OM4	OL4	OM5	OL5
TMSK1	OC1I	OC2I	OC3I	OC4I	I4O5I	-	-	-
TFLG1	OC1F	OC2F	OC3F	OC4F	I4O5F	-	-	-

11.2 Output Compare 1

- Event occurs when TCNT equals TOC1.
- Five external pins associated with event (OC1…OC5/PA7…PA3).
- Pins to control are selected by OC1M (OC1 mask register).
- The value of the controlled pins is determined by OC1D (OC1 data register).
- One event (OC1) can control five pins. All change at the same time.

Output Compare 1 Control

OC1My (OC1M)	1 = Enables control of pin PAy 0 = Disables control of pin PAy
OC1Dy (OC1D)	1 = Set pin PAy high on OC1 event 0 = Set pin PAy low on OC1 event

Output Compare 1 I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OC1M	OC1M7	OC1M6	OC1M5	OC1M4	OC1M3	0	0	0
OC1D	OC1D7	OC1D6	OC1D5	OC1D4	OC1D3	0	0	0

11.3 Output Compares 2…5

- Event occurs when TCNT equals TOCx.
- One external pin associated with each event (OC2/PA6, OC3/PA5, OC4/PA4, OC5/PA3).
- Control and value determined by associated OM and OL bits.
- Each event can control one pin.

Output Compare 2…5 Control

OMx	OLx	Level
0	0	Disconnected (not controlled)
0	1	Toggle
1	0	Low
1	1	High

Output Compare 2…5 I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TCTL1	OM2	OL2	OM3	OL3	OM4	OL4	OM5	OL5

11.4 Examples

11.4.1 Generating a Pulse Width (Output Compare 1)

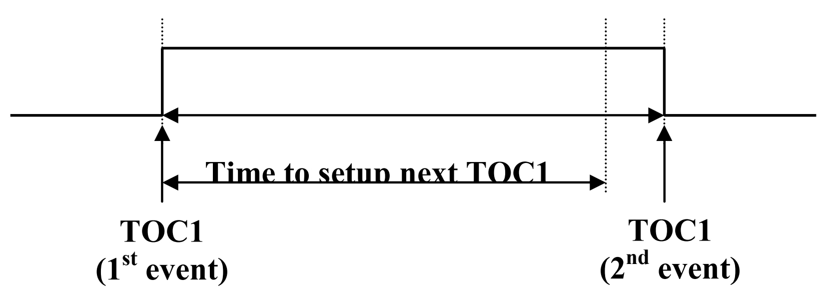
- Objective
    - Generate the signal shown below.

- Strategy
    - Use Output Compare 1 event.
    - Connect signal to an output pin controlled by OC1 (pick OC1/PA6).
    - Algorithm
        - Configure OC1/PA6 to go high on next event.
        - Clear OC1 flag.
        - Wait for OC1 event to occur.
        - Configure OC1/PA6 to go low on next event.
        - Set new OC1 event time.
        - Clear OC1 flag.
        - Wait for OC1 event to occur.
        - Disable OC1 control.

- Sample Source Code

  `ocpw1.c (CME11E9, simulator)`

- Observations
    - No delay between event time and signal change.
    - Limited only by time required by software to establish the next event time.



- Questions?
    - What is the minimum pulse duration that can be generated using this method?
    - What is the maximum pulse duration that can be generated using this method?

11.4.2 Generating a Pulse Width (Output Compare 2)

- Objective
    - Generate the signal shown below.

- Strategy
    - Use Output Compare 2 event.
    - Connect signal to an output pin controlled by OC2 (OC2/PA6).
    - Algorithm
        - Configure OC2/PA6 to toggle on event.
        - Set PA6 low.
        - Clear OC2 flag.
        - Wait for OC2 event to occur.
        - Set new OC2 event time.
        - Clear OC2 flag.
        - Wait for OC2 event to occur.
        - Disable OC2 control.

- Sample Source Code

  `ocpw2.c (CME11E9, simulator)`

- Observations
    - No delay between event time and signal change.
    - Limited only by time required by software to establish the next event time (shorter than previous example).

11.4.3 Generating a Pulse Width (Output Compare 1 and 2)

- Objective
    - Generate the signal shown below.

- Strategy
    - Use Output Compare 1 and 2 events.
    - Connect signal to an output pin controlled by OC1 and OC2 (OC2/PA6).
    - Algorithm
        - Configure OC1 to set signal high.
        - Configure OC2 to set signal low.
        - Set OC1 and OC2 times.
        - Clear OC1 and OC2 flags.
        - Wait for OC2 event to occur.
        - Disable OC1 and OC2 control.

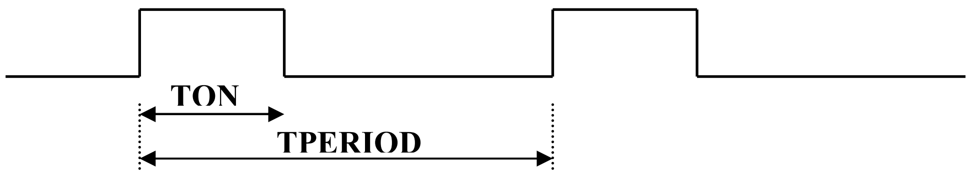
- Sample Source Code

    `ocpw12.c (CME11E9, simulator)`

- Observations
    - No delay between event time and signal change.
    - No limitation due to software. Only limitation is due to hardware.

11.4.4 Generating a PWM Signal (Output Compare 1 and 2)

- Objective
  - Generate the periodic signal shown below.
  - By changing (modulating) the pulse width, the average voltage can be controlled.
  - Pulse Width modulation (PWM) is a method for generating an effective analog voltage from a digital signal.



$$DutyCycle = \frac{TON}{TPERIOD}$$

$$V_{ave} = 5 \cdot DutyCycle$$

- Strategy
  - Use Output Compare 1 and 2 events.
  - Connect signal to an output pin controlled by OC1 and OC2 (OC2/PA6).
  - Algorithm
    - Configure OC1 to set signal high.
    - Configure OC2 to set signal low.
    - Set OC1 time.
    - Begin loop
      - Clear OC1 and OC2 flags.
      - Wait for OC1 event to occur.
      - Set OC2 and next OC1 times.
    - Repeat loop

- Sample Source Code

```
ocpwm.c (CME11E9, simulator)
```

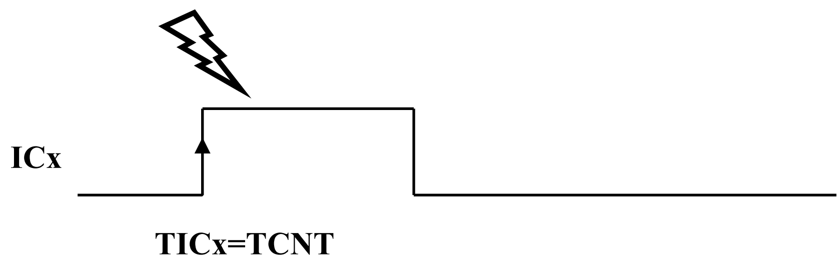
- Observations
  - No delay between event time and signal change.
  - Software must have sufficient time after OC1 event to update OC2 time (limitation on minimum on time).

- Questions?
  - What is the minimum period/pulse width that can be generated using this method?
  - What is the maximum period/pulse width that can be generated using this method?

12 Input Captures

12.1 Overview

- One external pin associated with each event.
- Event occurs when corresponding input signal generates specified edge (**Input** Capture).
- When event occurs, TCNT is stored in the corresponding TICx register (Input **Capture**).
- Stopwatch type event.



Input Capture Events (1…4)

Flag	Interrupt Enable	Control Registers	Pins
ICxF (TFLG1)	ICxI (TMSK1)	EDGxB,EDGxA (TCTL2) (TICx)	IC1…IC3,IC4 PA2…PA0,PA3

Input Capture Control

EDGxB	EDGxA	Edge
0	0	Disabled (not controlled)
0	1	Rising
1	0	Falling
1	1	Any

Input Capture I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TIC1(Hi)	15	14	13	12	11	10	9	8
TIC1(Lo)	7	6	5	4	3	2	1	0
TIC2(Hi)	15	14	13	12	11	10	9	8
TIC2(Lo)	7	6	5	4	3	2	1	0
TIC3(Hi)	15	14	13	12	11	10	9	8
TIC3(Lo)	7	6	5	4	3	2	1	0
TI4O5(Hi)	15	14	13	12	11	10	9	8
TI4O5(Lo)	7	6	5	4	3	2	1	0
TCTL2	EDG4B	EDG4A	EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A
TMSK1	-	-	-	-	I4O5I	IC1I	IC2I	IC3I
TFLG1	-	-	-	-	I4O5F	IC1F	IC2F	IC3F

12.2 Example – Measuring a Pulse Width

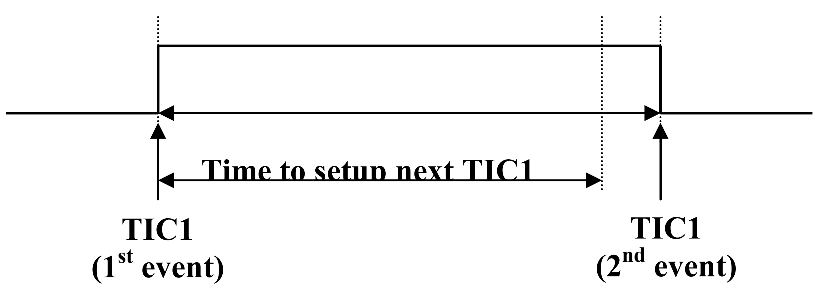
- Objective
    - Measure the pulse width of the signal shown below

- Strategy
    - Connect signal to an input capture pin (pick IC1/PA2).
    - Configure IC1/PA2 for rising edge.
    - Clear IC1 flag. Wait for IC1 event to occur.
    - Record start time.
    - Configure IC1/PA2 for falling edge.
    - Clear IC1 flag. Wait for IC1 event to occur.
    - Record stop time.
    - Calculate pulse width (ticks)
    - Disable IC1 control.

- Sample Source Code

```
icpw.c (CME11E9, simulator)
```

- Observations
    - No delay between detection of edge and recording of time.
    - Limited only by time required by software to prepare for the next event time.

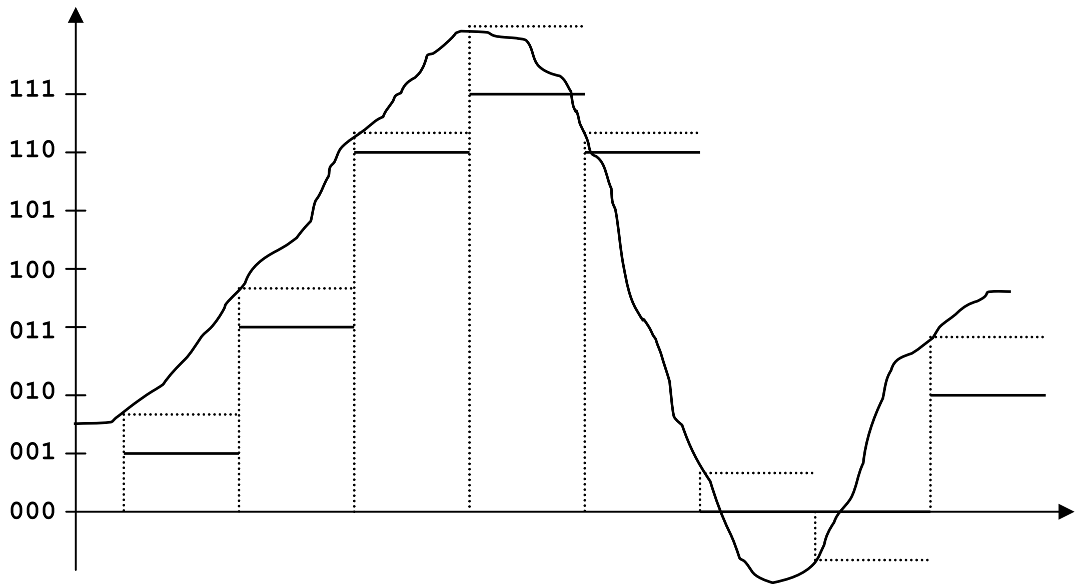


- Questions?
    - What is the minimum pulse width that can be measured using this method?
    - What is the maximum pulse width that can be measured using this method?

13 Analog-to-Digital Conversion

13.1 Overview

- An analog-to-digital converter (ADC) takes an analog signal (continuous value and time), samples it (discrete time), and converts the value into a digital (discrete value) representation.



- The voltage conversion range is determined by the V_{rh} and V_{rl} signals. On the EVBU, these signals are connected to +5 volts and 0 volts, respectively. The full-scale voltage range (V_{FS}) is the difference between the high and low reference voltages.

  Full-scale voltage range: $V_{FS} = (V_{rh} - V_{rl}) = 5$ volts

- The number of bits on the ADC determines how accurately the analog voltage can be represented. The 68HC11 has one 8-bit ADC. The step-size/resolution is the amount of voltage range associated with one count.

  Step size/resolution: $\Delta V = \left(\frac{V_{FS}}{2^N}\right) = \left(\frac{5}{256}\right) = 0.01953125$ volts/count

- The relationship between the analog value (V_{in}) and digital conversion (D_{adc}) is shown below.

  Digital conversion value: $D_{adc} = \left\lfloor \frac{(V_{in} - V_{rl})}{\Delta V} \right\rfloor = \left\lfloor \frac{V_{in}}{\Delta V} \right\rfloor$ counts

  Approximate analog value: $V_{in} \approx D_{adc} \cdot \Delta V$ volts

- Example:

  $V_{in} = 3.2$ volts $\qquad D_{adc} = \left\lfloor \frac{3.2}{0.01953} \right\rfloor = \lfloor 163.84 \rfloor = 163$ counts

  $D_{adc} = 163$ counts $\qquad V_{in} \approx 163 \cdot 0.01953 = 3.1836$ volts

13.2 Configuration

Analog-to-Digital Conversion Control

CD	CC	CB	CA	MULT=0	MULT=1 (ignore CB CA bits)
0	0	0	0	AN0→ADRx	AN0→ADR1
0	0	0	1	AN1→ADRx	AN1→ADR2
0	0	1	0	AN2→ADRx	AN2→ADR3
0	0	1	1	AN3→ADRx	AN3→ADR4
0	1	0	0	AN4→ADRx	AN4→ADR1
0	1	0	1	AN5→ADRx	AN5→ADR2
0	1	1	0	AN6→ADRx	AN6→ADR3
0	1	1	1	AN7→ADRx	AN7→ADR4
1	-	-	-	test signals	test signals

Analog-to-Digital Conversion I/O Registers

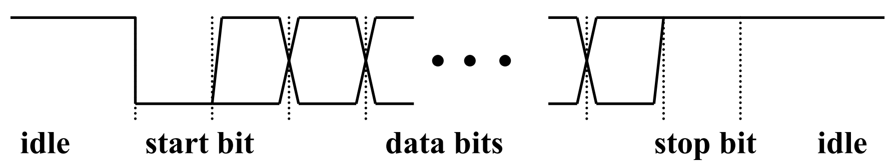
Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADCTL	CCF	0	SCAN	MULT	CD	CC	CB	CA
ADR1	7	6	5	4	3	2	1	0
ADR2	7	6	5	4	3	2	1	0
ADR3	7	6	5	4	3	2	1	0
ADR4	7	6	5	4	3	2	1	0
OPTION	ADPU	CSEL	IRQE	DLY	CME	0	CR1	CR0

- The ADC requires extra power to operate. To conserve power, the ADC on the 68HC11 is initially turned off. The ADPU bit turns on/off the analog-to-digital converter (0=off, 1=on).
- There is one 8-bit analog-to-digital converter. A 16 input multiplexer is used to select which signal to convert from 16 possible sources. Eight sources are from Port E. The other eight are for testing purposes.
- The conversion process is started by writing to the ADCTL register. One conversion takes 64 ticks (32 usec). Conversions are always done in sets of four. The conversion results are stored in the four ADRx registers.
- SCAN bit determines number of conversion sets (0=one set, 1=continuous)
- MULT bit determines number of signals to convert per set (0=one signal, 1=four signals)
- CD, CC, CB, CA bits determine which signal(s).
- CCF flag is set by hardware when an entire set of conversions is completed. The flag bit is cleared by writing to the ADCTL register.

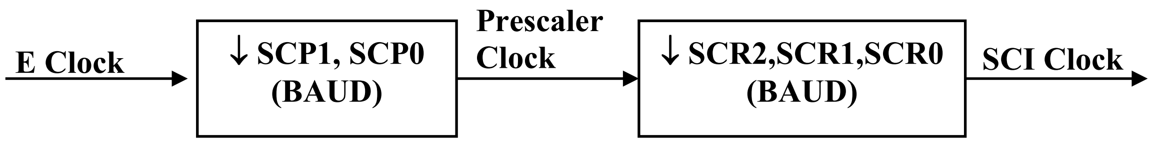
14 Serial Communication Interface (SCI)

14.1 Overview

- Serial communication is a standard interface used to transmit data one bit at a time across a single line.
- Because only one line is used to send the data, each bit is sent for a fixed time duration before the next bit is sent. On the receiver end, the line is checked at fixed time intervals to determine the bit value. The rate at which the bits are sent (line is checked) is called the BAUD rate. For example, a BAUD rate of 9600 means 9600 bits are sent every second.
- Bits are grouped into frames. Each frame has additional control bits (start, stop) for synchronization.



14.2 Baud Rate



Serial Communication Interface (SCI) Clock

SCP1	SCP0	↓Scale	Prescaler Rate [Hz]
0	0	1*16	125000
0	1	3*16	41667
1	0	4*16	31250
1	1	13*16	9600

SCR2	SCR1	SCR0	↓Scale	BAUD Rate [bps] (Prescaler = 9600)
0	0	0	1	9600
0	0	1	2	4800
0	1	0	4	2400
0	1	1	8	1200
1	0	0	16	600
1	0	1	32	300
1	1	0	64	150
1	1	1	128	75

Serial Communication I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
BAUD	-	-	SCP1	SCP0	-	SCR2	SCR1	SCR0
SCCR1	R8	T8	-	M	-	-	-	-
SCCR2	-	-	-	-	TE	RE	-	-
SCSR	TDRE	-	RDRF	-	-	-	-	-
SCDR	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0

14.3 Transmit/Receive Data

Serial Communication Interface (SPI)

	Enable	Flag	Data	Pins
Transmit	TE (SCCR2)	TDRE (SCSR)	M (SCCR1) (SCDR)	TxD (PD1)
Receive	RE (SCCR2)	RDRF (SCSR)	M (SCCR1) (SCDR)	RxD (PD0)

- The Transmit Enable (TE) bit must be set to a 1 to enable transmission of serial data.
- The Receive Enable (TE) bit must be set to a 1 to enable reception of serial data.
- The Mode (M) bit selects the character format
  0 = start bit, 8 data bits, 1 stop bit
  1 = start bit, 9 data bits, 1 stop bit
- The Serial Communication Data Register (SCDR) is used for both the transmit data register (write only) and the receive data register (read only).
- Transmit Data Register Empty (TDRE) flag set to a 1 by hardware when the SCDR register is empty (ready to send new data). Flag cleared by reading the SCSR register and then writing to the SCDR.
- Receive Data Register Full (RDRF) flag set to a 1 by hardware when the SCDR register is full (new data received). Flag cleared by reading the SCSR register and then reading from the SCDR.

15 Pulse Accumulator

15.1 Overview

- Accumulates (counts) number of pulses (edges).
- Two modes of operation:
  - Event counting – asynchronous counting (counts pulses)
  - Gated time counting – synchronous counting (counts time)

15.2 Configuration

Pulse Accumulator Events

Event	Flag	Interrupt Enable	Control Registers	Pins
Input Edge	PAIF (TFLG2)	PAII (TMSK2)	(PACTL) (PACNT)	PA7
Overflow	PAOVF (TFLG2)	PAOVI (TMSK2)	(PACTL) (PACNT)	PA7

Pulse Accumulator Control Register (PACTL)

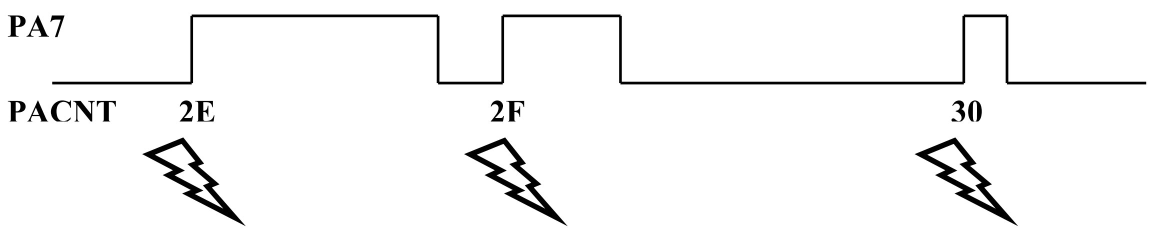
Bit	Name	Value	Meaning
PAEN	Pulse accumulator enable	0	disable
		1	enable
PAMOD	Pulse accumulator mode	0	event counting mode
		1	gated time accumulation mode
PEDGE	Pulse accumulator edge	0	falling edge (counting mode) active high (gated-time mode)
		1	rising edge (counting mode) active low (gated-time mode)

Pulse Accumulator I/O Registers

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PACTL	DDRA7	PAEN	PAMOD	PEDGE	-	-	-	-
PACNT	7	6	5	4	3	2	1	0

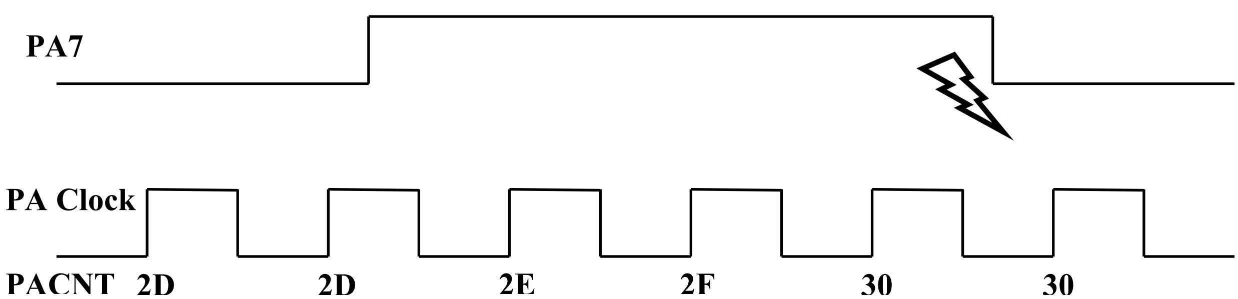
15.3 Event Counting Mode

- PAEN = 1, PAMOD = 0
- PAI event occurs on specified edge of PA7 (PEDGE bit)
- PACNT register increments on PAI event
- PAOV event occurs on overflow of PACNT register



15.4 Gated Time Accumulation Mode

- PAEN = 1, PAMOD = 1
- PAI event occurs on de-assertion of PA7 (PEDGE bit)
- PACNT register increments while PA7 asserted
- PACNT register increments on pulse accumulator clock edge (E-Clock/64)
- PAOV event occurs on overflow of PACNT register

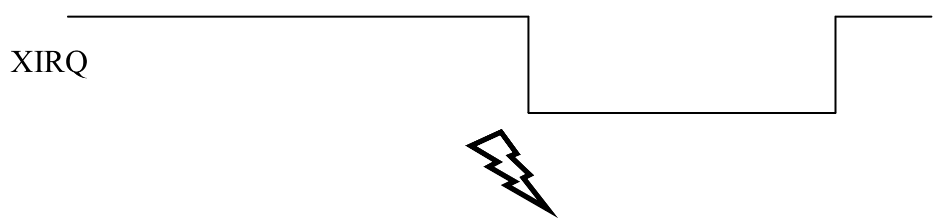


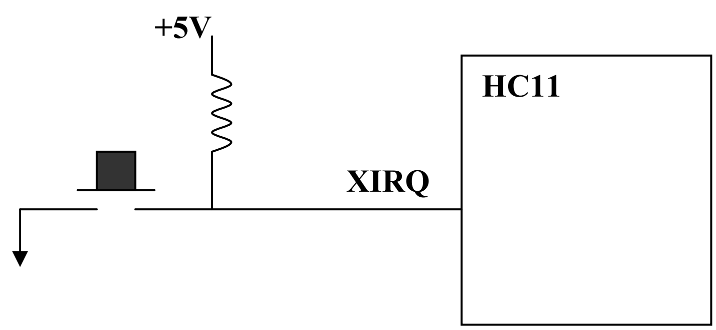
16 Interrupt Requests

- Occurs on specified signal level or edge

Interrupt Request (IRQ, XIRQ) Events

Event	Flag	Interrupt Enable	Control Registers	Pins
XIRQ	None	None	None	XIRQ
IRQ	None	None	IRQE (OPTION)	IRQ
IRQ	STAF (PIOC)	STAI (PIOC)	EGA (PIOC)	STAA





Appendix A: Sample Source Code

standard.c (DOS)

```
/* FILENAME:   standard.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      DOS application
 * DESCRIPTION:
 * Sample source code for standard I/O functions
 * This will not compile with the gcc 68HC11 compiler
 * for the EVBU due to insufficient memory
 */

#include <stdio.h>


int main(void)
{
        int   data=26;

        printf("Hello World\n");
        printf("sizeof(data) = %u\n",sizeof(data));
        printf("data = %d\n",data);
        printf("data = 0x%08X\n",data);

        return 0;
}
```

buffalo.c (CMM11E1, CME11E9, simulator)

```
/* FILENAME:   buffalo.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CMM11E1, CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for BUFFALO I/O functions
 * First part identical to standard I/O example
 */

#include <buffalo.h>


int main(void)
{
	int   data=26;
	char  value;
	int   counter;

	/* Same output as standard.c */
	puts("Hello World\n");
	puts("sizeof(data) = ");
	putint(sizeof(data));
	puts("\n");

	puts("data = ");
	putint(data);
	puts("\n");

	puts("data = 0x");
	puthex16(data);
	puts("\n");

	/* Example of BUFFALO I/O functions */

	/* Read and write characters */
	puts("Please press a key: ");
	value = getch();
	puts("\n");
	puts("You typed ");
	putch(value);
	puts("\n");

	/* Loops until character received */
	puts("Press any key to stop...");
	counter = 0;
	do {
		if (++counter == 1000) {
			counter = 0;
			puts(".");
		}
		value = input();
	} while (value==0);
	puts("\nYou typed ");
	putch(value);
	puts("\n");

	/* Read and write integers */
	puts("Please enter a number: ");
	counter = getint();
	puts("\nYou typed ");
	putint(counter);
	puts("\n");

	return 0;
}
```

int.c (CMM11E1, CME11E9, simulator)

```
/* FILENAME:   int.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CMM11E1, CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for integer/floating-point math comparison
 */

void func(int j)
{
}

int main(void)
{
        int j;

        for (j=0; j<100; j++)
                func(j);

        return 0;
}
```

float.c (CMM11E1, CME11E9, simulator)

```
/* FILENAME:   float.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CMM11E1, CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for integer/floating-point math comparison
 */

void func(float j)
{
}

int main(void)
{
        float j;

        for (j=0; j<100; j++)
                func(j);

        return 0;
}
```

tcnt.c (CMM11E1, CME11E9, simulator)

```
/* FILENAME:   tcnt.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:     CMM11E1, CME11E9, simulator (see comments)
 * DESCRIPTION:
 * Sample source code for free running counter
 * Used to measure how long instructions take to execute
 */

#include <buffalo.h>
#include <hc11e9.h>

void Wait(void) {
       puts("...Press any key..."); while ( input() == 0 ); puts("\n");
}

void DisplayTime(char *text, unsigned time)
{
       puts(text); putuint(time); puts(" ticks\n");
}

int main(void)
{
       unsigned start, stop;

       puts("File: "); puts(__FILE__); puts("\n");
       puts("TCNT timing demonstration.\n");
       puts("Press any key to begin...\n");
       getch();

       /* Continually display time */
       do {
              DisplayTime("TCNT = ",TCNT);
       } while ( input() == 0 );
       Wait();

       /* Measure how long it takes to update an I/O register */
       start = TCNT;
       PORTA = 0x23;
       stop = TCNT;
       DisplayTime("Start = ",start);
       DisplayTime("Stop = ",stop);
       DisplayTime("Time = ",stop-start);
       Wait();

       /* Measure how long it takes to transfer strings across serial port */
       /* !!! Timing does not behave properly on simulator !!! */
       start = TCNT;
       puts("1");
       stop = TCNT;
       DisplayTime("\nTime = ",stop-start);

       start = TCNT;
       puts("1234567890");
       stop = TCNT;
       DisplayTime("\nTime = ",stop-start);

       start = TCNT;
       puts("123456789012345678901234567890");
       stop = TCNT;
       DisplayTime("\nTime = ",stop-start);

       start = TCNT;
       puts("12345678901234567890123456789012345");
       stop = TCNT;
       DisplayTime("\nTime = ",stop-start);

       return 0;
}
```

tcntdelay.c (CMM11E1, CME11E9)

```
/* FILENAME:   tcntdelay.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CMM11E1, CME11E9
 * DESCRIPTION:
 * Sample source code for free running counter
 * Used to perform a 5 second delay
 */

#include <buffalo.h>
#include <hc11e9.h>


/* This function uses a polling loop to check
 * the free running counter. It will wait
 * approximately the number of passed ticks
 * before returning.
 */
void delay(unsigned ticks)
{
        unsigned start;

        start=TCNT;     /* Record starting time */
                        /* Calculate elapsed time and compare */
        while ((TCNT-start)<ticks);
}


int main(void)
{
        int i;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("TCNT 5 second delay.\n");

        /* Loop 5000 times - total delay of 5 second */
        for (i=0; i<5000; i++) {
                /* Delay approximately 1 msec */
                delay(2000);
        }

        return 0;
}
```

tcntmpw.c (CME11E9, simulator)

```
/* FILENAME:   tcntmpw.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CME11E9, simulator (tcntmpw_pa2.sti)
 * DESCRIPTION:
 * Sample source code for free running counter
 * Used to measure pulse width (PA2)
 */

#include <buffalo.h>
#include <hc11e9.h>


/* This function measures a positive pulse width
 * on PA2 and returns the time in ticks.
 */
unsigned MeasurePulseWidth(void)
{
        unsigned start, stop, width;  /* Stop watch variables */

        while ( (PORTA & PA2) == PA2);        /* Ensure start during low part */

        while ( (PORTA & PA2) == 0);  /* Wait until PA2 high */
        start = TCNT;                 /* Record start time */
        while ( (PORTA & PA2) == PA2);        /* Wait until PA2 low */
        stop = TCNT;                  /* Record stop time */
        width = stop - start;         /* Calculate pulse width (ticks) */

        return width;
}


int main(void)
{
        unsigned width;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Measuring pulse width on PA2...\n");

        width = MeasurePulseWidth();
        puts("Width = "); putuint(width); puts(" ticks\n");

        return 0;
}
```

tcntgpw.c (CME11E9, simulator)

```
/* FILENAME:   tcntgpw.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for free running counter
 * Used to generate pulse width (PA6)
 */

#include <buffalo.h>
#include <hc11e9.h>


/* This function generates a positive pulse width
 * on PA6. The width is passed as a parameter in ticks.
 */
void GeneratePulseWidth(unsigned width)
{
       unsigned start;                       /* Stop watch variables */

       PORTA |= PA6;                  /* Set PA6 high */
       start = TCNT;                  /* Record start time */
       while ( (TCNT-start) < width);        /* Wait */
       PORTA &= ~PA6;                 /* Set PA6 low */
}


int main(void)
{
       unsigned width = 100;

       puts("File: "); puts(__FILE__); puts("\n");
       puts("Generating "); putuint(width); puts(" tick pulse on PA6.\n");
       GeneratePulseWidth(width);

       return 0;
}
```

tofdelay.c (CMM11E1, CME11E9)

```
/* FILENAME:   tofdelay.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CMM11E1, CME11E9
 * DESCRIPTION:
 * Sample source code for timer overflow event
 * Used to perform a 5 second delay
 */

#include <buffalo.h>
#include <hc11e9.h>


int main(void)
{
        int i;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Timer overflow 5 second delay.\n");

        /* The free running counter overflows every 32.768 msec.
           If we wait for the TO event to occur 160 times it will
           correspond to a delay of approximately 5.1 seconds.
           The first TO event will not occur after 32.768 msec because
           we do not know exactly what time we entered the loop.
        */
        for (i=0; i<160; i++) {
                TFLG2 = TOF;    /* Clear flag */
                while ((TFLG2 & TOF) == 0);
        }

        return 0;
}
```

tof.c (CMM11E1, CME11E9, simulator)

```
/* FILENAME:   tof.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CMM11E1, CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for timer overflow event
 * Continuously displays free running counter
 * Displays "hit" whenever TO event is noticed
 */

#include <buffalo.h>
#include <hc11e9.h>

/* Required for usable simulation */
void Delay(void)
{
        unsigned start = TCNT;
        while ( (TCNT-start) < 2000 );
}

int main(void)
{
        char key;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Timer overflow event hit display (polling loop implementation).\n");
        puts("Press 'q' to quit.\n");
        puts("Press any key to begin...\n");
        getch();

        /* Polling loop */
        do {
                key = toupper(input());

                /* Check if TO flag set */
                if (TFLG2 & TOF) {
                        /* TO event occured */
                        TFLG2 = TOF;   /* Clear flag */
                        puts("Hit\n");
                }

                /* Display free running counter */
                puthex16(TCNT); puts("\n");

        } while (key!='Q');

        return 0;
}
```

toi.c (CMM11E1, CME11E9, simulator)

```
/* FILENAME:   toi.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:     CMM11E1, CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for timer overflow interrupt
 * Continuously displays free running counter
 * Displays "hit" whenever TO interrupt occurs
 */

#include <buffalo.h>
#include <hc11e9.h>


/* ISR function declaration syntax */
void TO_ISR(void) __attribute__((interrupt));


/* ISR function definition */
void TO_ISR(void)
{
        TFLG2 = TOF;           /* Clear flag */
        puts("Hit\n");
}

int main(void)
{
        char key;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Timer overflow event hit display (interrupt implementation).\n");
        puts("Press 'q' to quit.\n");
        puts("Press any key to begin...\n");
        getch();

        /* Setup ISR */
        __asm__("sei");                              /* Mask interrupts */
        *(unsigned *)0x00D1 = (unsigned)TO_ISR;      /* Assign TO PIV */
        TMSK2 |= TOI;                          /* Enable TO interrupt */
        TFLG2 = TOF;                           /* Clear flag */
        __asm__("cli");                              /* Unmask interrupts */

        /* Polling loop */
        do {
                key = toupper(input());

                /* Display free running counter */
                puthex16(TCNT); puts("\n");

        } while (key!='Q');

        /* Cleanup ISR */
        __asm__("sei");               /* Mask interrupts */
        TMSK2 &= ~TOI;         /* Disable TO interrupt */

        return 0;
}
```

tocntr.c (CMM11E1, CME11E9, simulator)

```
/* FILENAME:   tocntr.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CMM11E1, CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for timer overflow event
 * Implements 32.768 msec counter
 */

#include <buffalo.h>
#include <hc11e9.h>

unsigned CheckEvent(void)
{
        /* TO event configured to occur every 32.768 msec */
        if (TFLG2 & TOF)
        {
                TFLG2 = TOF;   /* Clear flag */
                return 1;
        }
        return 0;
}

int main(void)
{
        char key;
        unsigned event_counter=0;
        unsigned polling_counter=0;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Timer overflow event counter (polling loop implementation).\n");
        puts("Press 'e' for event counter.\n");
        puts("Press 'p' for polling-loop counter.\n");
        puts("Press 'q' to quit.\n");

        do {
                event_counter += CheckEvent();

                polling_counter++;

                key = toupper(input());
                switch (key) {
                case 'E':
                        putuint(event_counter); puts("\n");
                        break;
                case 'P':
                        putuint(polling_counter); puts("\n");
                        break;
                }
        } while ( key != 'Q' );

        return 0;
}
```

rticntr.c (CMM11E1, CME11E9, simulator)

```
/* FILENAME:   rticntr.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CMM11E1, CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for real-time interrupt event
 * Implements 4.096 msec counter
 */

#include <buffalo.h>
#include <hc11e9.h>

unsigned CheckEvent(void)
{
        /* RTI event configured to occur every 4.096 msec */
        if (TFLG2 & RTIF)
        {
                TFLG2 = RTIF;  /* Clear flag */
                return 1;
        }
        return 0;
}

int main(void)
{
        char key;
        unsigned event_counter=0;
        unsigned polling_counter=0;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Real-time interrupt event counter (polling loop implementation).\n");
        puts("Press 'e' for event counter.\n");
        puts("Press 'p' for polling-loop counter.\n");
        puts("Press 'q' to quit.\n");

        PACTL &= ~(RTR1 | RTR0);      /* Set RTI to 4.096 msec */

        do {
                event_counter += CheckEvent();

                polling_counter++;

                key = toupper(input());
                switch (key) {
                case 'E':
                        putuint(event_counter); puts("\n");
                        break;
                case 'P':
                        putuint(polling_counter); puts("\n");
                        break;
                }
        } while ( key != 'Q' );

        PACTL &= ~(RTR1 | RTR0);

        return 0;
}
```

ocpw1.c (CME11E9, simulator)

```
/* FILENAME:   ocpw1.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for output compares
 * Used to generate pulse width w/ OC1 (PA6)
 */

#include <buffalo.h>
#include <hc11e9.h>


/* This function generates a positive pulse width
 * on PA6. The width is passed as a parameter in ticks.
 */
void GeneratePulseWidth(unsigned width)
{
       TOC1 = TCNT + 200;            /* Set to start in 100 usec. */

       OC1M = OC1M6;                 /* Configure OC1/PA7 to go high on next event. */
       OC1D = OC1D6;

       TFLG1 = OC1F;                 /* Clear OC1 flag. */
       while ((TFLG1 & OC1F) == 0);  /* Wait for OC1 event to occur. */

       OC1D = 0;                     /* Configure OC1/PA7 to go low on next event. */
       TOC1 = TOC1 + width;          /* Set new OC1 event time. */

       TFLG1 = OC1F;                 /* Clear OC1 flag. */
       while ((TFLG1 & OC1F) == 0);  /* Wait for OC1 event to occur. */

       OC1M = 0;                     /* Disable OC1 control. */
}


int main(void)
{
       unsigned width = 100;

       puts("File: "); puts(__FILE__); puts("\n");
       puts("Generating "); putuint(width); puts(" tick pulse on PA6.\n");
       GeneratePulseWidth(width);

       return 0;
}
```

ocpw2.c (CME11E9, simulator)

```
/* FILENAME:   ocpw1.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for output compares
 * Used to generate pulse width w/ OC2 (PA6)
 */

#include <buffalo.h>
#include <hc11e9.h>


/* This function generates a positive pulse width
 * on PA6. The width is passed as a parameter in ticks.
 */
void GeneratePulseWidth(unsigned width)
{
        TOC2 = TCNT + 200;            /* Set to start in 100 usec. */

        TCTL1 |= OL2;                 /* Configure OC2/PA6 to toggle on event. */
        TCTL1 &= ~OM2;
        PORTA &= ~PA6;                /* Set PA6 low. */

        TFLG1 = OC2F;                 /* Clear OC2 flag. */
        while ((TFLG1 & OC2F) == 0);  /* Wait for OC2 event to occur. */

        TOC2 = TOC2 + width;          /* Set new OC2 event time. */

        TFLG1 = OC2F;                 /* Clear OC2 flag. */
        while ((TFLG1 & OC2F) == 0);  /* Wait for OC2 event to occur. */

        TCTL1 &= ~(OM2 | OL2);        /* Disable OC2 control. */
}


int main(void)
{
        unsigned width = 100;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Generating "); putuint(width); puts(" tick pulse on PA6.\n");
        GeneratePulseWidth(width);

        return 0;
}
```

ocpw12.c (CME11E9, simulator)

```
/* FILENAME:   ocpw12.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for output compares
 * Used to generate pulse width w/ OC1 & OC2 (PA6)
 */

#include <buffalo.h>
#include <hc11e9.h>


/* This function generates a positive pulse width
 * on PA6. The width is passed as a parameter in ticks.
 */
void GeneratePulseWidth(unsigned width)
{
	TOC1 = TCNT + 200;            /* Set to start in 100 usec. */

	OC1M = OC1M6;                 /* Configure OC1 to set signal high. */
	OC1D = OC1D6;
	TCTL1 |= OM2;                 /* Configure OC2 to set signal low. */
	TCTL1 &= ~OL2;

	TOC2 = TOC1 + width;

	TFLG1 = OC1F | OC2F;          /* Clear OC1 and OC2 flags. */
	while ((TFLG1 & OC2F) == 0);  /* Wait for OC2 event to occur. */

	OC1M = 0;                     /* Disable OC1 and OC2 control. */
	OC1D = 0;
	TCTL1 &= ~(OM2 | OL2);
}


int main(void)
{
	unsigned width = 100;

	puts("File: "); puts(__FILE__); puts("\n");
	puts("Generating "); putuint(width); puts(" tick pulse on PA6.\n");
	GeneratePulseWidth(width);

	return 0;
}
```

ocpwm.c (CME11E9, simulator)

```
/* FILENAME:   ocpwm.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CME11E9, simulator
 * DESCRIPTION:
 * Sample source code for output compares
 * Used to generate pwm signal w/ OC1 & OC2 (PA6)
 */

#include <buffalo.h>
#include <hc11e9.h>


/* This function generates a pulse width modulated signal
 * on PA6. The width and period are passed as a parameter in ticks.
 */
void GeneratePWM(unsigned width, unsigned period)
{
        unsigned num_pulses = 10;

        TOC1 = TCNT + 200;            /* Set to start in 100 usec. */
        TOC2 = TCNT;

        OC1M = OC1M6;                 /* Configure OC1 to set signal high. */
        OC1D = OC1D6;
        TCTL1 |= OM2;                 /* Configure OC2 to set signal low. */
        TCTL1 &= ~OL2;

        do {
               TFLG1 = OC1F | OC2F;           /* Clear OC1 and OC2 flags. */
               while ((TFLG1 & OC1F) == 0);  /* Wait for OC1 event to occur. */

               TOC2 = TOC1 + width;           /* Set OC2 and next OC1 times. */
               TOC1 = TOC1 + period;
        } while ( num_pulses-- );

        OC1M = 0;                     /* Disable OC1 and OC2 control. */
        OC1D = 0;
        TCTL1 &= ~(OM2 | OL2);
}


int main(void)
{
        unsigned width = 100, period = 300;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Generating PWM on PA6.\n");
        putuint(width); puts(" tick on time\n");
        putuint(period); puts(" tick period\n");
        GeneratePWM(width, period);

        return 0;
}
```

icpw.c (CME11E9, simulator)

```
/* FILENAME:   icpw.c
 * AUTHOR:     A. Blauch, GVSU
 * COMPATIBILITY:      CME11E9, simulator (icpw_pa2.sti)
 * DESCRIPTION:
 * Sample source code for input captures
 * Used to measure pulse width w/ IC1 (PA2)
 */

#include <buffalo.h>
#include <hc11e9.h>


/* This function measures a positive pulse width
 * on PA2 and returns the time in ticks.
 */
unsigned MeasurePulseWidth(void)
{
        unsigned start, stop, width;

        TCTL2 |= EDG1A;                       /* Configure IC1/PA2 for rising edge. */
        TCTL2 &= ~EDG1B;

        TFLG1 = IC1F;                 /* Clear IC1 flag. */
        while ((TFLG1 & IC1F) == 0);  /* Wait for IC1 event to occur. */

        start = TIC1;                 /* Record start time */
        TCTL2 |= EDG1B;                       /* Configure IC1/PA2 for falling edge. */
        TCTL2 &= ~EDG1A;

        TFLG1 = IC1F;                 /* Clear IC1 flag. */
        while ((TFLG1 & IC1F) == 0);  /* Wait for IC1 event to occur. */

        stop = TIC1;                  /* Record stop time */
        width = stop - start;         /* Calculate pulse width (ticks) */

        TCTL2 &= ~(EDG1A | EDG1B);    /* Disable IC1 control. */

        return width;
}


int main(void)
{
        unsigned width;

        puts("File: "); puts(__FILE__); puts("\n");
        puts("Measuring pulse width on PA2...\n");

        width = MeasurePulseWidth();
        puts("Width = "); putuint(width); puts(" ticks\n");

        return 0;
}
```

Appendix B: Sample Programming Problems

1. Write a program that sets the output signal on PD2 equal to the complement of the input signal on PD3. The program should loop continuously until the input signal on PE3 becomes high.

2. Write a program that generates a digital output signal that is the complement of a digital input signal (software implementation of a NOT gate). The program should loop continuously. You may decide which pin(s) to use.

3. Write a program that displays on HyperTerminal the alphabet in reverse order. The letters are to be displayed at a rate of one character every 1 sec.

4. Write a program that generates a 10 kHz, 60% duty cycle signal. The program should loop continuously. You may decide which pin(s) to use.

5. Write a program that generates a single 0.5 μsec pulse. The program should generate one pulse and then exit. You may decide which pin(s) to use.

6. Write a program that will measure the period of a square wave (50% duty cycle) input signal. The period is to be displayed on HyperTerminal in μsec. The program should perform one measurement and then exit. You may decide which pin(s) to use.

7. Write a program to determine the time-shift between two digital input signals. The time-shift should be measured between the rising edges and displayed on HyperTerminal in μsec. The program should perform one measurement and then exit. You may decide which pin(s) to use.

8. Write a program that calculates the average (DC) value of an analog input signal. You must take a measurement every millisecond. The average must be calculated over a 100 msec interval and displayed on HyperTerminal in counts. The program should perform one measurement and then exit. You may decide which pin(s) to use.

9. Write a program to determine the turn-on threshold voltage for the 68HC11 digital input circuitry. The turn-on voltage is to be displayed on HyperTerminal in counts. The program should perform one measurement and then exit. You may decide which pin(s) to use.

10. Write a program the compares two analog input signals, V_1 and V_2. A digital output signal must be set to a 1 if $V_1 > V_2$, and to a 0 if $V_1 <= V_2$ (software implementation of a comparator circuit). The program should loop continuously. You may decide which pin(s) to use.