ECSE-4790 Microprocessor Systems

Motorola 68HC12 User's Manual

Lee Rosenberg
Electrical and Computer Systems Engineering
Rensselaer Polytechnic Institute

Revision 1.1
8/10/00

Table of Contents: page

1. **Introduction:**

The Motorola 68HC12 is a 16-bit microprocessor descended from the 68HC11. The design has a number of major improvements over the 6811 and several new features that are not found on the 6811. The biggest change is the expansion from an 8-bit bus to a full 16-bit bus for both the data and address. Other improvements include an increase in the number of A/D converter registers, Timer output compare and input capture pins, and I/O ports. Also added is a second SCI connector and a new interrupt, called a Key Wakeup interrupt.

This manual is intended to provide a brief introduction to the 68HC12 and how to program it in C using the Introl C compiler 4.00. This manual is intended primarily for those people who are already familiar with the Motorola 68HC11. This manual also assumes that the reader has basic familiarity with the C programming language.

2. **Basic Programming Notes:**

There are 3 header files that must be included with any code written for the 68HC12 using the Introl C compiler. These are:

HC812A4.H - This file contains all the register declarations for the 6812.

INTROL.H - This file contains several function declarations needed by Introl to compile the program.

DBUG12.H - This contains the information need to call the D-Bug12 routines and to handle interrupts. Omitting this file will result in the calls to the D-Bug12 routines being flagged as errors by the compiler.

Your main function must be of the format: void __main(). The two (2) underscores before main are necessary, as that is the format that Introl uses to recognize the main function of the program.

3. D-Bug12:

D-Bug12 is the monitor program for the 6812 EVB. This is similar to the BUFFALO monitor used on the 6811. Unlike the BUFFALO monitor, the D-Bug12 monitor is a series of C functions that are stored in an EPROM on the EVB. These functions can be called by the user to handle both I/O and several of the common C language ANSI functions.

All calls to the D-Bug12 routines follow the same format. The format is:

```
DB12->"routine name";
```

The "DB12->" is used as a cast pointer that allows the compiler to reference the EPROM for the different routines. The routine name is just the name of the routine and any parameters that are being passed to the function. If the function returns a value to the program the return value can be assigned to a variable. This is done as follows:

```
temp=DB12->"routine name";
```

This assigns the return value of the routine to a variable named temp. As always the variable must be declared in the program.

It is important to note that if you do not include the "DB12->" with the function call the compiler will return an error message. The error message that is returned is that the function does not exist. Putting the "DB12->" before the function name will solve this problem.

D-Bug12 Functions

Readers interested in a more in-depth explanation of the D-Bug12 routines are referred to Motorola Document an1280a, "Using the Callable Routines in D-Bug 12" (available on the web at http://www.ecse.rpi.edu/Courses/CStudio/appnotes/).

getchar

This function will get a single character of input from the user.

Function Prototype: int getchar(void);
Return Value: This returns the character from the keyboard in hexadecimal ASCII.

printf

This will display a string of characters to the screen.

Function Prototype: int printf(char *s);
Return value: The number of characters that were transmitted.

NOTE: The Introl 4.0 compiler has an error in this function. The first parameter in the list is not printed properly. There are workarounds for some cases that are given in examples in class handouts. In any case, simple strings without variables will work without problems.

To display a variable, the variable is represented using %y in the printf statement, where y is chosen from the table below to match the variable type. To display a signed decimal integer stored in a variable num, the function call would look like:

```
DB12->printf("This is the value of num: %d", num);
```

d, i	int, signed decimal number
o	int, unsigned octal number
x	int, unsigned hexadecimal number using a-f for 10-15
X	int, unsigned hexadecimal number using A-F for 10-15
u	int, unsigned decimal
c	int, single character
s	char *, display from a string until '\0'
p	void *, pointer

putchar

This will display a single ASCII character on the screen.

Function Prototype: int putchar(int);
Return Value: The character that was displayed.

GetCmdLine

This function is used to read in a line of data from the user and store it in an array. Each character that is entered is echoed back to the screen using a call to the putchar function. Only printable ASCII characters are accepted by the function with the exception of carriage return and backspace.

Function Prototype: int GetCmdLine(char *CmdLineStr, int CmdLineLen);
Return Value: An error code of NoErr.

The location where data that is read in is stored and the number of characters that are to be read, are determined by CmdLineStr and CmdLineLen respectively.

CmdLineStr is a char array that is created by the programmer. This is where the input line from the user is stored.

CmdLineLen is the length of the string to be read in. A total of CmdLineLen -1 characters may be entered by the user before the GetCmdLine function exits. The user may also use a carriage return to exit before the value of CmdLineLen has been reached.

Backspace may be used to delete unwanted characters from the line that the user entered. The character will be erased from the screen and the memory array.

isxdigit

This routine determines if c is a member of the set 0..9, a..z and A…Z

Function prototype: int isxdigit(int c);
Return Value: If c is part of the set the function returns true (1), if c is not a part of the set the function returns a false (0).

toupper
This routine is used to convert lower case letters to upper case letters.
Function Prototype: int toupper(int c);
Return Value: The uppercase value of c. If c is already an uppercase letter it returns c.

isalpha
This routine determines if c is a member of the set a…z and A…Z
Function Prototype: int isalpha(int c);
Return Value: If c is part of the set it returns true (1), if it is not a part of the set it returns a false (0).

strlen
This routine determines the length of the string that is passed in as a parameter. The string must be null terminated.
Function Prototype: unsigned int strlen(const char *cs);
Return Value: The length of the string pointed to by cs.

strcpy
This routine makes a copy of string two to string one. The string to be copied must be null terminated.
Function Prototype: char* strcpy(char *s1, char *s2);
Return Value: A pointer to s1.

out2hex
This outputs an 8-bit number on the screen as two hexadecimal numbers.
Function Prototype: void out2hex(unsigned int num);
Return Value: None

out4hex
This outputs a 16-bit number on the screen as four hexadecimal numbers.
Function Prototype: void out4hex(unsigned int num);
Return Value: None

SetUserVector
This routine is used for handling Interrupt Service Routines and will be described in the section on interrupts.

Notes:

The printf, putchar, out2hex, and out4hex routines do not generate carriage returns or line feeds when they are called. To do this you must include \n\r in either a putchar or a printf statement.

4. 68HC12 Hardware:

This section explains the operation of the hardware on the 68HC12. It includes the I/O ports, the A/D converter and the timer functions. Hardware interrupts are explained in the next section. For more information on this material, refer to Motorola document MC68HC812A4TS/D, "Motorola 68HC12 Technical Summary".

Ports:

All port names are of the format _H12PORTx, where x is the capital letter of the port you are trying to access. i.e. Port A is _H12PORTA.

Ports A and B are used as the address bus for the 68HC12. They are not usable as I/O by the programmer. Port A is the high order byte and port B is the low order byte.

Ports C and D are used as the data bus for the 68HC12. They are not usable for I/O by the programmer. Port C is the high order byte and port D is the low order byte.

Port E is used to generate control signals needed to access the external memory. As a result the programmer can not use it for I/O. Port E pin 1 is used as the input for the IRQ and Port E pin 0 is used as the input for the XIRQ.

Port F is used to control chip selects for the external memory and other chips. The programmer can not use it as I/O.

Port G is a 6-bit general purpose I/O port. The direction of the port is controlled by _H12DDRG. When an _H12DDRG bit is set to 0, the port pin is an input and when it is set to 1 it is an output.

Port H is an 8-bit general purpose I/O port. The direction of the port is controlled by the _H12DDRH. When _H12DDRH bit is set to 0, the port pin is an input and when it is set to 1 it is an output.

Port J is an 8-bit general purpose I/O port. The direction of the port is controlled by the _H12DDRJ. When _H12DDRJ bit is set to 0, the port pin is an input and when it is set to 1 it is an output.

Port S is used for the SCI and the SPI. It can also be used for general I/O if the SCI or SPI are not being used. Bits 0 and 1 are SCI0. These are used as the interface to the terminal and can not be used as I/O. Bits 2 and 3 are SCI1 and bits 4-7 are the SPI. These can be used as general I/O if SCI1 and the SPI are not being used. The direction of the port is controlled by _H12DDRS.

Port T is used for the timer interrupts and the pulse accumulator. If the interrupts are not being used then Port T can be used for general I/O. The port direction is controlled by _H12DDRTT. Note: The two T's are not a typo.

Port AD is used exclusively as the input to the A to D converter. It can not be used for any other I/O.

Ports G, H, and Port E pin 0 have optional pull-up resistors inside. These are controlled by the _H12PUCR register. To enable the pull-up resistor for Port H write a one to bit 7, for Port G write a one to bit 6, for Port E

write a one to bit 4 of _H12PUCR. To disable the pull-up resistors to a particular port, write a 0 to the appropriate bit.

A to D Converter:

This section covers the basic function of the A/D converter. The A/D converter uses a dedicated port, port AD, for its inputs. There are 8 A/D converter channels on the HC12 and, unlike the 6811, there are also 8 A/D registers allowing 8 simultaneous readings. The implementation of the A/D converter allows for two different methods of operating the A/D converter. This section covers the polling method. The interrupt based operation is explained in the section on interrupts.

The A/D converter conversion sequence consists of either 4 or 8 conversions and can convert either one channel or multiple channels. In scan mode operation, polling based operation, the flag is set after the conversions have been completed, signaling the completion of the A/D cycle.

Before running the A/D converter, the system must be initialized by the user. There are 4 registers that are used to control the A/D Converter. These are _H12ADTCTL2, _H12ADTCTL3, _H12ADTCTL4, and _H12ADTCTL5.

_H12ADTCTL2 contains several of the A/D Converter enable bits. Bit 7 is the A/D power up (ADPU). When this is set to one the A/D converter is enabled, when it is zero then the A/D converter is disabled. Bit 1 is the A/D converter interrupt enable (ASCIE). The interrupt is enabled when the bit equals one and disabled when the bit is set to zero. In scan mode this bit is set to 0, to disable interrupts. Bit 0 is the interrupt flag (ASCIF) which is not used in the polling version of the A/D converter.

_H12ADTCTL2:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
ADPU	AFFC	AWAI	unused	unused	unused	ASCIE	ASCIF

_H12ADTCTL3 should always be set to 0x00. This is used to control several actions that are related to how the A/D converter operates in background debug mode. As the D-Bug12 monitor is being used, the background debug mode is not being used and these features should be disabled.

_H12ADTCTL4 is used to select the sample time of the A/D converter and to set the prescalar for the clock.

_H12ADTCTL4:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
unused	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0

There are four different sample times available for the A/D converter. The sample time is selected by setting the value of SMP1 and SMP0. The different sample times that can be used are listed in table 1.

Table 1: Sample Times

SMP1	SMP0	Sample Time
0	0	2 A/D clock periods
0	1	4 A/D clock periods
1	0	8 A/D clock periods
1	1	16 A/D clock periods

The prescalar that is used by the A/D converter is determined by the value of PRS0-PRS4. The clock input to the prescalar is an 8 MHz clock. This allows

for an A/D conversion frequency of 500 kHz to 2 MHz. The different prescalar values are listed in Table 2.

Table 2: Prescalar Values

Prescale Value	Divisor
00000	Do not use
00001	/4
00010	/6
00011	/8
00100	/10
00101	/12
00110	/14
00111	/16
01xxx	Do not use
1xxxx	Do not use

_H12ADTCTL5 is used to select the conversion mode, which channels are to be converted and to initiate the conversions. The conversion sequence is started by any write made to this register. If a write is made to this register while a conversion sequence is in progress, the conversion is aborted and the SCF and CCF flags are reset.

_H12ADTCTL5:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
unused	S8CM	SCAN	MULT	CD	CC	CB	CA

S8CM is used to select between making 4 conversions, when the bit is set to zero, and 8 conversions, when the bit is set to one.

SCAN is used to select between performing either a single conversion or multiple conversions. If SCAN is set to zero, then a single conversion will be run and the flag will then be set. If SCAN is set to 1, then the A/D converter will run continuous conversions on the A/D channels.

MULT determines whether the conversion is run on a single channel or on multiple channels. When MULT zero, the A/D converter runs all the conversions on a single channel, which is selected by CD,CC,CB, and CA. When MULT is one, the conversion is run on several different channels in the group specified by CD,CC,CB, and CA. The possible channel combinations are in Table 3.

Table 3: A/D Converter Settings

S8CM	CD	CC	CB	CA	Channel Signal	Result in ADRx if MULT = 1
0	0	0	0*	0*	AN0	ADR0
0	0	0	0*	1*	AN1	ADR1
0	0	0	1*	0*	AN2	ADR2
0	0	0	1*	1*	AN3	ADR3
0	0	1	0*	0*	AN4	ADR0
0	0	1	0*	1*	AN5	ADR1
0	0	1	1*	0*	AN6	ADR2
0	0	1	1*	1*	AN7	ADR3
0	1	0	0*	0*	Reserved	ADR0
0	1	0	0*	1*	Reserved	ADR1
0	1	0	1*	0*	Reserved	ADR2
0	1	0	1*	1*	Reserved	ADR3
0	1	1	0*	0*	V RH	ADR0

0	1	1	0*	1*	V RL	ADR1
0	1	1	1*	0*	(V RH + V RL)/2	ADR2
0	1	1	1*	1*	TEST/Reserved	ADR3
1	0	0*	0*	0*	AN0	ADR0
1	0	0*	0*	1*	AN1	ADR1
1	0	0*	1*	0*	AN2	ADR2
1	0	0*	1*	1*	AN3	ADR3
1	0	1*	0*	0*	AN4	ADR4
1	0	1*	0*	1*	AN5	ADR5
1	0	1*	1*	0*	AN6	ADR6
1	0	1*	1*	1*	AN7	ADR7
1	1	0*	0*	0*	Reserved	ADR0
1	1	0*	0*	1*	Reserved	ADR1
1	1	0*	1*	0*	Reserved	ADR2
1	1	0*	1*	1*	Reserved	ADR3
1	1	1*	0*	0*	V RH	ADR4
1	1	1*	0*	1*	V RL	ADR5
1	1	1*	1*	0*	(V RH + V RL)/2	ADR6
1	1	1*	1*	1*	TEST/Reserved	ADR7

Stared (*) bits are “don’t care” if MULT = 1 and the entire block of four or eight channels make up a conversion sequence. When MULT = 0, all four bits (CD, CC, CB, and CA) must be specified and a conversion sequence consists of four or eight consecutive conversions of the single specified channel.

_H12ADSTAT is used to determine the status of the conversion process. Unlike most of the registers in the HC12 this is a 16-bit register. The Sequence Complete Flag (SCF) is used to signal the completion of the conversion cycle. When SCAN = 0, the setting of the SCF signals the completion of the cycle, when SCAN = 1, it signals the completion of the first conversion cycle.

_H12ADSTAT:

bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8
SCF	unused	unused	unused	unused	CC2	CC1	CC0
CCF7	CCF6	CCF5	CCF4	CCF3	CCF2	CCF1	CCF0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

CC2-CC0 are the conversion counter. They are the pointer for the conversion cycle and reflect which result register will be written to next.

CCF7-CCF0 are the Conversion Complete Flags (CCF) for the individual A/D channels. When the conversion sequence for a channel has been complete the flag is set. The flags can be cleared by reading the A/D register for the channel and by reading the _H12STAT register.

The results of the A/D conversions are stored in the A/D converter result registers. These registers are called _H12ADR0H through _H12ADR7H. When converting multiple channels, the destination register used to store the results of each channel that is converted is listed in Table 3. When converting a single channel, the results are in _H12ADR0H - _H12ADR3H for a 4 conversion sequence and _H12ADR0H - _H12ADR7H for an eight conversion sequence.

Sample Code

This code turns on the A/D converter while disabling the A/D interrupt. It then performs 4 conversions on channel 0 of the A/D converter and displays the result that is stored in the result registers to the screen.

```
_H12ADTCTL2=0x80;   // turn on ATD and off the interrupt
_H12ADTCTL3=0x00;   // don't stop at breakpoints
_H12ADTCTL4=0x43;   // Set prescalar (/8) & sample time 8 periods

_H12ADTCTL5=0x00;   // check AN0, 4 conversions and stop
while(!(_H12ADTSTAT & 0x8000)); // wait for flag to be set
DB12->out2hex(_H12ADR0H);       // Display A/D result registers
DB12->printf("\n\r");
DB12->out2hex(_H12ADR1H);
DB12->printf("\n\r");
DB12->out2hex(_H12ADR2H);
DB12->printf("\n\r");
DB12->out2hex(_H12ADR3H);
```

Timer Functions:

This section covers the operation of the timer module when it is used for non-interrupt based operations, such as output compare and input capture functions. Interrupt based timer features are discussed in chapter 5.

The basics of timer module operation:
There are several basic features of timer module that apply to both the input capture and output compare functions, as well as interrupt driven Timer functions.

In order to make use of any timer based operations, the timer module must first be enabled. This is done by setting _H12TSCR to 0x80. This will set the Timer Enable bit (TEN), which then enables all timer operations.

_H12TSCR:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TEN	TSWAI	TSBCK	TFFCA	unused	unused	unused	unused

The Timer uses a counter, also called a free running counter, that is incremented by one every clock pulse. The free running counter on the 68HC12 can be accessed by the user if needed. The free running counter is a 16-bit value that is stored in the register _H12TCNT. It can be read at anytime, but can not be written to by the user.

Port T is used as I/O pins for the timer input capture and timer output compare. Each pin can serve as either an input capture or output compare pin. The function of the pin is selected by the state of the _H12TIOS register. Each bit in _H12TIOS corresponds to a pin of Port T. When the bit is set to 0, the pin is used as an input capture, when the bit is set to a one, the pin is used as an output compare. Any combination of input captures and output compares can be selected by the user.

_H12TIOS:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
pin 7	pin 6	pin 5	pin 4	pin 3	pin 2	pin 1	pin 0

The data for the input captures and output compares are stored in _H12TC0 to _H12TC7. These are 16 bit registers. For input capture operations the value of the free running counter will be latched into the register when the input capture is triggered. For output compare operations the value in the register is used to trigger an action.

The 68HC12 also allows the user to assign pull-up resistors to the timer module inputs. This is done by writing a one to TPE in _H12TMSK2. This will enable the pull-ups. A zero will disable them.

_H12TMSK2:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TOIE	unused	TPE	TDRB	TCRE	PR2	PR1	PR0

Timer Output Compare

The output compare on the 68HC12 is very similar to that of the 68HC11. The user selects an action to occur when the output compare is triggered and the time at which the action occurs. The processor will then carry out this action accordingly.

The first step is to select which channel(s) will be used. This is done by using the _H12TIOS register as explained above. Having done this, the next step is to select when the output compare will trigger. This is done by writing a value to the _H12TCx register(s) that correspond to the channel(s) that was selected as an output compare in _H12TIOS.

It is then necessary to determine what action will occur when the output compare is triggered. There are several different actions that are possible, which one occurs is determined by the values in _H12TCTL1 and _H12TCTL2. These registers contain the control bits for each channel, OMn and OLn. The effect that the different values have are listed in Table 4.

Table 4: Output compare actions

OMn	OLn	Action
0	0	Timer disconnected from output logic
0	1	toggle Ocn output line
1	0	clear Ocn output line to 0
1	1	set Ocn output line to 1

_H12TCTL1:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4

_H12TCTL2:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0

Having determined the action taken on a successful match, the next step is to make sure timer interrupts are disabled. This is done by writing 0x00 to _H12TMSK1. Lastly the timer module is enabled, as shown above, allowing the output compare to trigger.

It is important to note when using the output compare that when an output compare action is triggered the corresponding bit of Port T will automatically be set as an output, regardless of the state of DDRTT.

Sample Code

This code sets up an output compare and causes the pin output to toggle when the interrupt is triggered.

```
_H12TIOS=0xFF;          // set up the channels as output compare
_H12TMSK1=0x00;         // no hardware interrupts
_H12TCTL1=0x5A;         // set OC7, OC6 for toggle/ OC5,OC4 clear
_H12TCTL2=0x5F;         // set OC3, OC2 for toggle/ OC1,OC0 set
_H12TC0=0x0000;         // set different trigger times
_H12TC1=0x2000;
_H12TC2=0x4000;
_H12TC3=0x6000;
_H12TC4=0x8000;
_H12TC5=0xA000;
```

```
_H12TC6=0xC000;
_H12TC7=0x9000;

_H12TSCR=0x80;          // turn on the timer
```

Output Compare 7

Output Compare 7 has a special feature that makes it very powerful. Output Compare 7 allows the programmer to change the state of any of the output compare pin, without changing the operation that is performed by that Output Compare. This is particularly useful for generating pulsewidth modulated signals. Anyone interested in this particular application should refer to the LITEC manual for more detailed information on pulsewidth modulation.

The method for using output compare 7 to control the other output compares is set up as follows. Just as with any other output compare operation, the time at which OC7 is triggered must be stored in _H12TOC7. The next step is to select which channels will be controlled by OC7 and what will occur when OC7 triggers. This is done using the _H12OC7M and _H12OC7D registers.

_H12OC7M is used to select which channels are controlled by OC7. Writing a one to a bit in _H12OC7M assigns control of the corresponding channel to OC7. The data that is output by the channel is stored in _H12OC7D. When OC7 is triggered, for each bit that is set in _H12OC7M, the corresponding data bit in _H12OC7D is written to the output compare pin.

A successful OC7 event can be used to cause the “free running” counter to be reset. This is done by writing a one to TCRE in _H12TMSK2. Note if you write a one and set the OC7 event for $0000 the free running counter will stay at $0000. Similarly if you set OC7 for $FFFF, the Timer Overflow Flag will never be set. (See Timer Overflow Interrupt.)

_H12OC7M:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
OC7M7	OC7M6	OC7M5	OC7M4	OC7M3	OC7M2	OC7M1	OC7M0

_H120C7D:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
OC7D7	OC7D6	OC7D5	OC7D4	OC7D3	OC7D2	OC7D1	OC7D0

_H12TMSK2:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TOIE	unused	TPE	TDRB	TCRE	PR2	PR1	PR0

Sample Code
This code simply shows how to setup OC7.

```
_H12TC7=0x4000;         // Set up the time when the OC triggers
_H12OC7M=0x01;          // select channel 0
_H12OC7D=0x00;          // cause channel 0 to output a low when
                        // triggered
```

Timer Compare Force Register

This is a special register that allows the programmer to cause an output compare to trigger. Writing to bit n in this register causes the action which is programmed for output compare n to occur immediately. This is the same as if a successful comparison had just taken place with the TCn register.

_H12CFORC:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
FOC7	FOC6	FOC5	FOC4	FOC3	FOC2	FOC1	FOC0

Input Capture

The operation of the input capture is similar to that of the output compare. The first step is to set up the appropriate Port T bits as input capture pins using _H12TIOS.

Having done that, it is then necessary to select how the input capture will be triggered. This is done using _H12TCTL3 and _H12TCTL4. Each channel has two control bits EDGxB and EDGxA, which determine which edge triggers the input capture. The different configurations of these bits are in Table 5.

Table 5: Input Capture selects

EDGxB	EDGxA	Configuration
0	0	Capture disabled
0	1	Capture on rising edge
1	0	Capture on falling edge
1	1	Capture on any edge.

_H12TCTL3:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
EDG7B	EDG7A	EDG6B	EDG6A	EDG5B	EDG5A	EDG4B	EDG4A

_H12TCTL4:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B	EDG0A

Just as with the output compare operation, the timer interrupts must be disabled and the timer module must be enabled. The values that are read in by the input capture are stored in the appropriate _H12TCx register.

Sample Code

This code captures the time of _H12TCNT when a switch is pressed by the user.

```
_H12TMSK1=0x00;   // turn off interrupts
_H12TIOS=0x00;    // Set up Port T for input capture
_H12TCTL3=0x5A;   // IC7, IC6 rising edge, IC5, IC4 falling edge
_H12TCTL4=0x5F;   // IC3, IC2 rising edge, IC1, IC0 any edge
_H12TSCR=0x80;    // turn on the timer
```

5. Interrupt Service Routines Overview:

Overview:

The 68HC12 provides a wide array of interrupts that can be used by the programmer. Some of these are similar to interrupts found on the 68HC11, such as the RTI and timer overflow. Others are new to the 68HC12 such as the key wakeups and A/D converter interrupts. This section will cover all the different ISRs and how they operate.

Interrupts on the 68HC12 are controlled in part through the D-Bug12 monitor program. The SetUserVector routine in the D-Bug12 monitor is used to program the ISR vector table that tells the processor where the different ISRs are located in memory.

When using a particular interrupt the ISR must be assigned to the interrupt in the beginning of the program. Each interrupt has an address offset to the interrupt vector table base address, which is stored in the D-Bug12 header file. When assigning the interrupt, SetUserVector is called and the name of the interrupt as well as the ISR name are passed as parameters. This will store the address of the ISR in the vector table.

For example: If you are using the Real Time Interrupt (RTI) and have it call an ISR called RTIInt when it is triggered, the code would look like:

```
DB12->SetUserVector(RTI, RTIInt);
```

The list below is all of the interrupts and their mnemonics:

AtoD	A to D converter interrupt
PAEdge	Pulse Accumulator Edge triggered
PAOvf	Pulse Accumulator Overflow triggered
TimerOvf	Timer Overflow
Timer7	Timer 7
Timer6	Timer 6
Timer5	Timer 5
Timer4	Timer 4
Timer3	Timer 3
Timer2	Timer 2
Timer1	Timer 1
Timer0	Timer 0
RTI	Real Time Interrupt
IRQ	IRQ interrupt
XIRQ	XIRQ interrupt

The format of the ISR is show below. All ISRs follow this format.

```
__mod2__ void RTIInt()
{
// your code here
}
```

It also must include a function prototype of the format:

```
__mod2__ void RTIInt();
```

Note: There are 2 underscores both before and after the mod2.

Interrupt Priority:

Interrupts on the 68HC12 do not all occur simultaneously. Rather there is a hierarchy of priority for the interrupts. The default priority order is:

1) Reset
2) COP Clock Monitor Fail Reset
3) COP Failure Reset
4) Trap
5) SWI
6) XIRQ
7) IRQ
8) RTI
9) Timer0
10) Timer1
11) Timer2
12) Timer3
13) Timer4
14) Timer5
15) Timer6
16) Timer7
17) TimerOvf
18) PAOvf
19) PAEdge
20) SPI
21) SCI0
22) SCI1
23) AtoD
24) PortJKey
25) PortHKey

The priority of these can be changed by using _H12HPRIO register. The first six interrupts are unmaskable and can not have their priority changed. The other interrupts are all maskable and may have their priority changed. An interrupt may be made the highest priority interrupt by writing its address value to _H12HPRIO. The address values are listed below in hex for each interrupt:

IRQ:	F2
RTI:	F0
Timer0:	EE
Timer1:	EC
Timer2:	EA
Timer3:	E8
Timer4:	E6
Timer5:	E4
Timer6:	E2
Timer7:	E0
TimerOvf:	DE
PAOvf:	DC
PAEdge:	DA
SPI:	D8
SCI0:	D6
SCI1:	D4
AtoD:	D2
PortJKey:	D0
PortHKey:	CE

Interrupt Service Routines example code and explanations

This section includes explanations of all the different interrupts and sample code of the function.

Real Time Interrupt:

Operation
The operation of the RTI is controlled by _H12RTICTL. Bit 7 is the Real Time Interrupt Enable (RTIE). Writing a one to this bit will enable the RTI. The rate at which the RTI is triggered is determined by the Real Time Interrupt Rate (RTR). The different rates are listed in Table 6.

Table 6: RTI rate

RTR2	RTR1	RTR0	Period___
0	0	0	off
0	0	1	1.024ms
0	1	0	2.048ms
0	1	1	4.096ms
1	0	0	8.196ms
1	0	1	16.384ms
1	1	0	32.768ms
1	1	1	65.536ms

_H12RTICTL:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
RTIE	RSWAI	RSBCK	unused	RTBYP	RTR2	RTR1	RTR0

After the interrupt is triggered the ISR must clear the flag. This is done by writing a one to the Real Time Interrupt Flag (RTIF) in _H12RTIFLG register.

_H12RTIFLG:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
RTIF	unused	unused	unused	unused	unused	unused	unused

Sample Code
This is a simple program to count the number of RTI interrupts that have occurred.

```
__mod2__ void RTIInt();                  // function prototype

int Timecount;                           // global variable

void __main()
{
      DB12->SetUserVector(RTI,RTIInt);   // set up interrupt vector

      TimeCount = 0;

      _H12RTICTL=0x87;                   // set up the RTI for 65.536ms
      while(1)
      {
      DB12->out2hex(Timecount);
      }
}
```

```
__mod2__ void RTIInt()                  // interrupt service routine
{
      Timecount++;
      _H12RTIFLG=0x80;                  // clear the flag
}
```

Timer Overflow Interrupt:

Operation
The Timer Overflow Interrupt (TOI) functions by generating an interrupt every time the "free running" counter overflows. The free running counter is a 16 bit value and is constantly running in the background when the timer is enabled.

The TOI is set up in the _H12TMSK2 register by writing a one to Timer Overflow Interrupt Enable (TOIE). After this has been done the Timer can be enabled. Whenever the interrupt is triggered the ISR will be called and executed. The ISR must clear the flag by writing a one to the Timer Overflow Flag in the _H12TFLG2 register.

_H12TMSK2:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TOIE	unused	TPE	TDRB	TCRE	PR2	PR1	PR0

_H12TFLG2:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TOF	unused	unused	unused	unused	unused	unused	unused

Sample Code
This code calls an interrupt every time the Timer Overflow Interrupt occurs.

```
      __mod2__ void TimerOvfInt(); // function prototype

      void __main()
      {
      DB12->SetUserVector(TimerOvf, TimerOvfInt);

      _H12TMSK2=0x80;              // enable interrupt
      _H12TSCR=0x80;               // enable the timer
      while(1)                     // idle loop
      {
      }
}

__mod2__ void TimerOvfInt()        // Timer Overflow ISR
{
      DB12->printf("Overflow interrupt");
      _H12TFLG2=0x80;              // clear the flag
}
```

Pulse Accumulator Edge Triggered Interrupt:

Operation
This section describes how to set up the Pulse Accumulator for edge triggered operation. The Pulse Accumulator is enabled by setting the Pulse Accumulator Enable (PAEN) in _H12PACTL to one. There are two other control bits in _H12PACTL, PAMOD and PEDGE.

When PAMOD equals zero the pulse accumulator is in event counter mode, when it is one it is in gated time accumulation mode. PEDGE has different effects based on the state of PAMOD.

When PAMOD equals zero:
If PEDGE equals 0, then falling edges on the pulse accumulator input pin (Port T bit 7) causes the count to be incremented.
If PEDGE equals 1, then rising edges on the input cause the count to be incremented.

When PAMOD equals one:
If PEDGE equals 0, when the pulse accumulator input pin goes high it enables an internal clock which is connected to the pulse accumulator and the trailing falling edge on the pulse accumulator input sets the PAIF flag. The internal clock used to increment the pulse accumulator is 8MHz/64.

If PEDGE equals one, when the pulse accumulator input pin goes low it enables an internal clock which is connected to the pulse accumulator and the trailing rising edge on the pulse accumulator input sets the PAIF flag. The internal clock used to increment the pulse accumulator is 8MHz/64.

The timer must be enable to use these since the clock generated is based on the timer prescaler.

CLK1 and CLK0 are used to control the clock rate at which the pulse accumulator is incremented. The different options are listed in Table 7.

Table 7: Pulse Accumulator Clock Rates

CLK1	CLK0	Selected clock
0	0	timer prescaler
0	1	8MHz clock
1	0	8MHz/256 clock
1	1	8MHz/65536 clock

PAI is the Pulse Accumulator Edge triggered interrupt enable. This must be set to 1 to enable edge triggered interrupts.

_H12PACTL:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
unused	PAEN	PAMOD	PEDGE	CLK1	CLK0	PAOVI	PAI

After setting PACTL, the hardware will trigger an interrupt whenever the correct edge is detected at Port T. The ISR must clear the flag by writing a one to the Pulse Accumulator Interrupt Flag (PAIF) in _H12PAFLG.

_H12PAFLG:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
unused	unused	unused	unused	unused	unused	PAOVIF	PAIF

The value of the pulse accumulator is stored in _H12PACNT.

Sample Code

This code detects the edge and triggers an interrupt.

```
__mod2__ void PAEdgeInt();    // function prototype

void __main();
{
      DB12->SetUserVector(PAEdge, PAEdgeInt);

      _H12PACTL=0x55;         // set the pulse accumulator - rising edge
      while(1)                // wait
      {

      }
}

__mod2__ void PAEdgeInt()     // Pulse Accumulator ISR
{
      DB12->printf("Pulse Accum triggered \n");
      _H12PAFLG=0x01;         // clear the flag
}
```

Pulse Accumulator Overflow Triggered Interrupt:

Operation
This operation triggers an interrupt every time the pulse accumulator overflows. Whenever the pulse accumulator overflows from 0xFFFF to 0x0000, the ISR will trigger. The set up of this interrupt is very similar to the edge triggered interrupt, although PAMOD and PAEDGE have no effect on the interrupt. Most of the other settings are the same as for the edge triggered operation of the Pulse Accumulator. However instead of setting PAI to one, for Overflow operation PAOVI is set to one.

The user must clear the flag in the ISR by writing a 1 to PAOVIF in the _H12PAFLG register.

_H12PACTL:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
unused	PAEN	PAMOD	PEDGE	CLK1	CLK0	PAOVI	PAI

_H12PAFLG:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
unused	unused	Unused	unused	unused	unused	PAOVIF	PAIF

Sample Code
This code simply displays when the interrupt is triggered.

```
__mod2__ void PAOvfInt();     // function prototype

void __main()
{
      DB12->SetUserVector(PAOvf, PAOvfInt);

      _H12PACTL=0x46;         // set for pulse accumulator overflow
      while(1)                // wait
      {
      }
}

__mod2__ void PAOvfInt()      // Pulse Accumulator ISR
{
      DB12->printf("triggered");
      _H12PAFLG=0x02;         // clear the flag
}
```

Output Compare Interrupt:

Operation

The output compare interrupt calls an ISR every time a successful output compare is detected. The output compare is setup like the non- interrupt based output compare, with the major difference being that the action which is to occur on a successful compare does not need to be specified. Instead, _H12TMSK1 is used to determine which channel(s) will be used to generate an interrupt(s). Each bit in _H12TMSK1 corresponds to a different output compare channel. When the interrupt is generated, the ISR for that channel is called and executed. The flag must be cleared by writing a one to the bit _H12TFLG1 register that corresponds to the channel which triggered the interrupt.

_H12TMSK1:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I

_H12FLG1:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F

Sample Code

This code calls the ISR when the Output Compare 0 generates an interrupt.

```
__mod2__ void Timer0Int();    // function prototype

void __main()
{
      DB12->SetUserVector(Timer0, Timer0Int);

      _H12TIOS=0xFF;          // select output compare
      _H12TMSK1=0x01;         // enable the interrupt on pin 0
      _H12TC0=0x8000;         // set the value to be compared against
      _H12TSCR=0x80;          // enable the timer

      while(1)                // wait
      {
      }
}

__mod2__ void Timer0Int()     // Output Compare ISR
{
      DB12->printf("timer int");
      _H12TFLG1=0x01;         // clear the flag
}
```

Input Capture Interrupt:

Operation

Like the output compare interrupt, the input capture interrupt is very similar to the non-interrupt based input capture. The _H12TIOS and _H12TCTL3 and _H12TCTL4 are set up the same as for the non-interrupt input capture. The difference is that _H12TMSK1 is used to determine which channel(s) will be used to generate an interrupt(s). Each bit in _H12TMSK1 corresponds to a different input capture channel. When the interrupt is generated, the ISR for that channel is called and executed. The flag must be cleared by writing a one to the bit in the _H12TFLG1 register that corresponds to the channel which triggered the interrupt.

_H12TMSK1:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I

_H12FLG1:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F

Sample Code

This code calls the ISR when the input capture generates an interrupt.

```
__mod2__ void Timer0Int();    // function prototype

void __main()
{
      DB12->SetUserVector(Timer0, Timer0Int);

      _H12TIOS=0x00;          // select input capture
      _H12TMSK1=0x01;         // enable the interrupt on pin 0
      _H12TCTL3=0x5A;         // IC7, IC6 rising edge, IC5, IC4 falling edge
      _H12TCTL4=0x5F;         // IC3, IC2 rising edge, IC1, IC0 any edge
      _H12TSCR=0x80;          // enable the timer

      while(1)                // wait
      {
      }
}

__mod2__ void Timer0Int()     // Output Compare ISR
{
      DB12->printf("timer int");
      _H12TFLG1=0x01;         // clear the flag
}
```

A to D Converter Interrupt:

Operation
This is similar to the non-interrupt based A/D converter. However instead of having to poll the flag to determine when the A/D cycle has been completed, an interrupt is generated when the conversion is completed.

In order to make use of the A/D interrupt the ASCIE bit in _H12ADTCTL2 must be set to 1. This way, when the conversion is completed the interrupt will be triggered. The flag is cleared by writing a one to ASCIF in _H12ADTCTL2. The remainder of the operation is the same as the non-interrupt based A/D converter.

_H12ADTCTL2:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
ADPU	AFFC	AWAI	unused	unused	unused	ASCIE	ASCIF

Sample Code
This sample code calls the A/D interrupt when the A/D cycle is completed.

```
      __mod2__ void AtoDInt();     // Function prototype

      void __main()                // main program
      {
      DB12->setUserVector(AtoD, AtoDInt); // set the vector address

      _H12ADTCTL2=0x82;       // turn on ATD and on the interrupt
      _H12ADTCTL3=0x00;       // don't stop at breakpoints
      _H12ADTCTL4=0x43;       // Set prescalar (/8) & sample time 8 periods
      _H12ADTCTL5=0x00;       // check AN0
      while(1)                // infinite loop
      {
      }
      }

__mod2__ void AtoDInt()      // A/D ISR
{


      DB12->out2hex(_H12ADR0H);     // print contents of registers
      DB12->printf("\n\r");
      DB12->out2hex(_H12ADR1H);
      DB12->printf("\n\r");
      DB12->out2hex(_H12ADR2H);
      DB12->printf("\n\r");
      DB12->out2hex(_H12ADR3H);
      _H12ADTCTL2=0x83;             // reset the A/D converter
      _H12ADTCTL5=0x00;
}
```

IRQ Interrupt:

Operation

The IRQ is used to generate external interrupts. There are two basic modes of operation for the IRQ. One is falling edge triggered, the other is low level detection. The mode of the IRQ is controlled by _H12INTCR. IRQEN is used to turn on the IRQ. When this bit is set to 1 the IRQ is enabled, when it is set to zero the IRQ is disabled. IRQE determines whether low level or edge triggered operation will be used. When this bit is 0, the IRQ will use low level detection. When it is 1 the IRQ will be falling edge triggered. The IRQ is automatically cleared by the hardware in the 68HC12.

There are a few differences between the IRQ on the HC11 and the HC12. Unlike the HC11, the IRQ is not time protected. IRQEN may be written to and read from at any time. The value of IRQE, however, may only be written once in the program.

_H12INTCR:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
IRQE	IRQEN	DLY	unused	unused	unused	unused	unused

Sample Code

This is sample code that triggers the IRQ every time a falling edge is detected. This assumes there is a switch of some sort on the IRQ input to generate the falling edge.

```
__mod2__ void IRQInt();      // function prototype

void __main()
{

      DB12->SetUserVector(IRQ, IRQInt);   // set up the vector

      _H12INTCR=0xC0;         // set up the IRQ for falling edge triggered

      while(1)                // infinite loop
      {

      }
}

__mod2__ IRQInt()             // IRQ ISR
{
      DB12->printf("IRQ triggered");      // display a message
}
```

Port H Key Wakeup Interrupt:

Operation

This is a new feature that was incorporated into the HC12. This generates an interrupt when the appropriate edge is detected on the input to the port. This is ideal for use with a keypad, or can be used as extra IRQ lines for the HC12.

Each bit of Port H can be used to generate a Key Wakeup interrupt when a falling edge is detected. It is important to note that even though each bit can generate an interrupt independently of the others, the same interrupt will be called regardless of which bit triggered it. The Key Wakeup Interrupt for each individual bit is enabled using _H12KWIEH. Writing a 1 to a bit in the register will enable the corresponding bit of Port H to generate an interrupt when a falling edge is detected. The flag bits for the Key Wakeup interrupt are located in _H12KWIFH. Multiple flags can be set at the same time, although software must be written in order to determine which flags have been set. The flags are cleared by writing a one to the flag bits that have been set. It is a good idea to clear the flags before the Key Wakeup Interrupt is enabled to prevent any false triggers.

_H12KWIEH:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

_H12KWIFH:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Sample Code

This code will trigger a Port H Key Wakeup interrupt when a falling edge is detected on any bit of Port H.

```
__mod2__ void KeyH(void);     //function prototype

void __main(void)
{
      DB12->SetUserVector(PortHKey,KeyH); // Set Vector address

      _H12KWIFH=0xFF;         // make sure flags aren't set.
      _H12KWIEH=0xFF;         //Enable all key wakeups for port H

      while(1)
      {
      }
}

__mod2__ void KeyH(void)      // Port H Key Wakeup ISR
{
      _H12KWIFH=_H12KWIFH;    // clear the flags
}
```

Port J Key Wakeup Interrupt:

Operation
The Port J key Wakeup Interrupt is a more powerful version of the Port H Key Wakeup Interrupt. Unlike the Port H Key Wakeup, Port J can be set to trigger on either a rising edge or a falling edge input. This adds to the flexibility of the key wakeup but also adds to the complexity.

The selection of which bits will be used to generate interrupts is controlled by _H12KWIEJ. As with port H, when a bit is set to 1, the Key Wakeup for that channel is enabled. Also as with the Port H Key Wakeup, while each bit can cause an interrupt, they all call the same interrupt service routine.

_H12KWIEJ:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

_H12KWIFJ:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Before enabling the Key Wakeup Interrupt it is necessary to set which edge will trigger the interrupt and to select to use either the pull-up or pulldown resistors on the input.

Which edge will be used to trigger an interrupt is determined by the setting of _H12KPOLJ. Writing a zero to a bit of _H12KPOLJ makes that channel falling edge triggered. Writing a one makes the channel rising edge triggered.

_H12KPOLJ:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

The pull-up and pulldown resisters are controlled by _H12PUPSJ and _H12PULEJ. _H12PUPSJ selects between using pull-up resisters and pulldown resisters. When a bit in _H12PUPSJ is set to one, the channel has a pull-up resistor. When the bit is zero, there is a pulldown resistor on the input. This MUST be set before the pull-up/pulldown resistors are enabled. _H12PULEJ is used to enable the pull-up or pulldown resistor. Writing a one enables the pull-up or pulldown, while writing a zero will disable it.

_H12PULEJ:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

_H12PUPSJ:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

As with the Port H Key Wakeup, the flags are cleared after a Port J Key Wakeup Interrupt is triggered. This is done by writing a one to the bits in _H12KWIFJ that have been set.

Sample Code
This code sets up the Port J Key Wakeup for falling edge operation on all 8 channels and enables the pull up resistors for all the bits.

```
__mod2__ void KeyJ(void);     // function prototype

void __main(void)
{
      DB12->SetUserVector(PortJKey, KeyJ); // assign the vector address

      _H12KPOLJ=0x00;         // falling edge sets flag
      _H12KWIFJ=0xFF;         // clear any flags that may be set
      _H12PUPSJ=0xFF;         // pull up
      _H12PULEJ=0xFF;         // pull up enabled all bits
      _H12KWIEJ=0xFF;         // Enable all bits of J for keypad

      while(1)                // Infinite loop
      {
      }
}

__mod2__ void KeyJ(void)      // function prototype
{
      _H12KWIFJ=_H12KWIFJ;    // clear the flag
}
```