868HC11 Laboratory Manual

N. Natarajan

Chapter 1

Not required

Chapter 2

Introduction to Looping

2.1 Objective

To become familiar with elementary loops and simple input/outputs functions.

2.2 Simple Input/Output

One of the basic functionality provided by any operating system is input/output routines. BUFFALO provides several useful functions for performing input/output. In this lab, we will look at two output functions provided by BUFFALO. Unlike in high level languages, functions in machine language are known by their addresses. The two functions we will be using are in ROM at locations `$FFB8` and `$FFBB`[1]. Rather than use these hard to remember and hard to recognize numbers, it is customary to give them meaningful names. Unless you have a good reason to do otherwise, it is best to use the name suggested by the vendor, in this case Motorola. The 'official' names for these functions are `OUTA` and `OUT1BYT` respectively. In assembly language, we make the connection between a name (technically known as a label) and a value using the EQU command as shown

```
OUTA          EQU     $FFB8
OUT1BYT       EQU     $FFBB
```

NOTE: Labels should be written starting from column 1. If a line does not have a label, it should start with a space or a tab or a comment character.

[1]We will indicate HEX values with the prefix $. However, data that you would be entering in BUFFALO, as part of memory modify or register modify will be shown without the prefix $ although it will be understood that the numbers are written in HEX.

2.2.1 The function `OUTA`

The function `OUTA` will transmit whatever is in register A over the serial communication line that is connected to the PC. What the PC does with this value depends on the terminal program that is used to communicate with the HC11. Under normal circumstances, the terminal program will interpret the value as an ASCII code and display the corresponding character on the screen.

Exercise: Power up HC11 and at the BUFFALO prompt try the following and write down what you see.

1. Issue the command RM (for register modify) and press the space bar till you see the **A** register. Enter the value 31 and press enter as shown below:

   ```
    >rm
   P-AAAA Y-AAAA X-AAAA A-AA B-AA C-D0 S-004A
    P-AAAA
    Y-AAAA
    X-AAAA
    A-AA 31

    >
   ```

   Now execute the command

   ```
    CALL FFB8
   ```

2. Repeat with the **A** register modified with the following values: `32, 33, 34, 21, 22, 23, 24, 25, 41, 42 43 44`

If you are using the simulator, turn on the log feature (by pressing both the shift keys). If you are working with a real HC11, you can copy and paste the contents of the terminal screen.

2.2.2 The function `OUT1BYT`

This is a more involved output function. To start with, the value to be printed must be in memory. If the value is in a register you will have to store it in memory first. Next, the value in the **X** register should be the address where the value is stored. Thus this function should be told 'where' and not 'what'. When you call this function, the function will send *two* characters to the PC. If the terminal program interprets these two characters as ASCII codes and displays the corresponding two characters, then the display would be the value written in HEX. *In addition, the function will increment the value in the* **X** *register.* This is useful when displaying a series of memory locations.

Exercise:

1. Using the MM command (memory modify) enter the following values in memory locations `3000, 3001`, $\cdots$: `30 31 32 33 41 42 43 44`. Verify the values using the memory dump, MD, command.

2. Using RM, the register modify command, change the value in the **X** register to `3000`

3. Execute the command `CALL FFBB` and write down what the output was and also the value in the **X** register after the command is executed.

4. Repeat the the command `CALL FFBB` and write down the output and the new value in the **X** register.

5. Repeat the last part until you have performed 7 calls to `$FFBB`.

2.3 Branching

Conditional branching in HC11 is controlled by the state of one or more hardware flags. The state of a flag depends on the most recently executed instruction that affects the flag. Thus, if your branching depends on the result of some instruction, then it is your responsibility to make sure that none of the instructions between the instruction you are interested in and the branching instruction affects the flags that control the branching instruction. Thus, it is a good idea to follow the instruction that sets the flag by the branching instruction. In this lab, we will use the following conditional branch instructions:

BEQ	Branch if the Z flag is set
BNE	Branch if the Z flag is not set

Now, the `Z` flag is set after most instructions if the result of the instruction is a *zero*; or else it is cleared, i.e. not set. The two most important instructions that are often used to set/clear the flag are

CMP	Perform a subtraction and discard the answer. However, set the flags.
TST	Same as CMP except subtract the number zero

Thus, after the `CMP` command, the `Z` flag would be set if the two values that are compared are equal. Similarly, the `TST` command will compare a value (memory or register) with zero and set the Z flag if the value is zero.

Exercise:

1. Using the HC11 reference book, identify 5 instructions that do *not* affect the **C** flag, but affects some other flag.
2. Using the HC11 reference book, can you identify any instruction that does *not* affect the **V** flag, but affects some other flag?
3. Using the HC11 reference book, identify 5 instructions that always clears the **C** flag.
4. Using the HC11 reference book, identify an instruction that always sets the **C** flag.

2.4 Looping

2.4.1 Counting loops

This is by far the simplest and most used loop structure. In a counting loop, we perform a specific operation a given number of times. A counter controls how many times the loop is executed. The counter could be stored in memory or kept in a register. If it is kept in a register, it is your responsibility to make sure that the register is not inadvertently changed either by your code or by some third party function that you call. If you decide to use a register, your best bet is to use either the **B** register or **Y** register. The structure of the loop is as follows:

1: Initialize the counter to the number of times the loop is to be performed

2: Perform any other initializations

3: Test if the counter is zero. If so quit the loop

4: Perform the desired task

5: Perform any re-initializations

6: Decrement the counter

7: Go back to 3

8: Come here when you quit the loop

Note that there are two places where the code jumps to. One to (3) and the other to (8). When writing the code in assembly language, we would need two labels. In the code that follows, I have used the labels `FOO` and `BAR`.

Exercises:

1. Assemble the following code in your PC, transfer the S19 file to HC11, run the program and write down the output of the program.

```
;Name:
;email:
;date:
;
OUTA         EQU     $FFB8

             ORG $2100
             LDAB    #$9 ; USING REGISTER B AS A COUNTER
             LDAA    #$31 ; OTHER INITIALIZATION
FOO          TSTB        ; SUBTRACT ZERO FROM B
             BEQ BAR     ; QUIT IF THE Z FLAG IS SET, I.E. B=0
             JSR OUTA    ; DO THE TASK
             INCA        ; RE-INITIALIZE
             DECB        ; DECREMENT THE COUNTER
             BRA  FOO    ; GO BACK

BAR
             SWI
```

2. Modify the above program so that the program prints the upper case letters `A` to `Z`. Clearly indicate the changes you made.

3. The following function is equivalent to the function shown above[2].

```
Code Deleted
```

Verify that `CALL 2100` and `CALL 2200` produces the same output. Explain why this is so, and how and why the loop terminates.

2.4.2 One, two! One, two! And through and through ... Marching through memory

Often loops are combined with marching through memory and operating on consecutive memory location. In this case, the **X** (and/or **Y**) register is initialized

[2]You can have as many functions as you want in the same file. Make sure that when you change the ORG, the functions do not overlap. You can determine this by looking at the LST file

to a starting memory address. Inside the loop, memory is accessed using `IND,X` addressing mode. This operates on memory whose address is computed using the value in **X** register. At the bottom of the loop, **X** is incremented, so that next time around the loop, the operation is performed on the next memory location[3]. The following program shows prints using `OUTA` the values stored in 9 consecutive locations starting from location `$3000`. Note we are using FCB which is the assembly language equivalent of memory modify.

```
Code Deleted
```

Run the above program using `CALL 2300`[4]. Modify the above program as shown below and explain what the program does. Do you see why we do the `DEX` before we access memory?

```
Code Deleted
```

Now for some other useful examples. Explain what each of them does. Run the programs and using memory dump, verify that your explanation is correct. Each of the functions start with an `ORG` command.

```
Code Deleted
```

The following program prints the sixteen values stored in the consecutive location starting from $3020. To make the output more user friendly, each number is followed by a comma and a space.

```
Code Deleted
```

[3]In some cases the memory has to be accessed in the reverse order. In this case, **X** starts at the end, and at the bottom of the loop, **X** is decremented.

[4]Don't forget to assemble it and then transfer the S19 file to the HC11!

Chapter 3

Functions and bit manipulations

3.1 Objective

To become familiar with bit level operations and writing functions. This lab also illustrates the use of random numbers for testing functions.

3.2 What you should do

You will be writing several functions in this laboratory exercise. You should have only one file and you should add the new function at the end of your older functions. Also, you will be adding items to the data section. You should **not** delete earlier data items. Your main code will be changing. You should insert your new main code before the older one, so that the most recent main code will start immediately after the `ORG` statement. Try not to delete any code from your file.

3.3 String outputs

In the last lab, we saw how to write a single byte as an ascii character. To write a string, it is tedious to load A with one character at a time and then calling `OUTA` each time. A better approach is to put all the characters in consecutive memory locations and print them all in a loop. To do this, we need two pieces of information, where to start and where to end. The common approach to such situation is to specify where to start and use a special value, known as sentinel, to indicate the end. Some of you may have used special values such as zero, one, or 9999. It is entirely up to the programmer, but for character strings, the three most often used sentinels are zero (also known as ASCII-Z string), 26 (also known as CONTROL-Z string, or old DOS string), 4 (EOT string).

The programmers of BUFFALO use EOT string and you have one of two choices: rewrite BUFFALO routine and use some other sentinel, or use 4 as the sentinel and remember to place it after each string. The rest of the lab assumes that you will use the EOT string. Two functions that BUFFALO provides for printing strings are `OUTSTRG` at location `$FFC7` and `OUTSTRGO` at location `$FFCA`. The difference between them is that the former will print the string on a new line, while the latter will continue the string from wherever the cursor happens to be. Both these functions must be told where the string is. You do this by loading the **X** register with the starting address where the string is stored before calling the function. To enter strings in memory, we use FCC directive. The label associated with the FCC will be automatically `EQU`ated to the starting address of the string. Type the following code and verify that `OUTSTRG` does indeed print a string.

```
;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;
OUTSTRG      EQU $FFC7
OUTSTRGO     EQU $FFCA

; program section. set origin to $2100
        ORG $2100
        LDX #ABOUTME    *STARTING ADDRESS OF THE STRING
        JSR OUTSTRG
        SWI

; data section. set origin to $3000
        ORG $3000
ABOUTME FCC /Hello, my name is ===your name =====/
        FCB 4 ;dont forget this
;;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;
```

3.4 Writing your first function

Before you write your first function, you should observe some standard conventions. Your programs should always start at location $2100, or as specified by your TA. Your data should always start at location $3000 or as specified by your TA. A small amount of data can be stored starting at location $0000 (Page 0), but you do not have too much space, 30 bytes or so. It is a matter of taste whether you write the data section first or the program section first. You can not mix and match. I personally prefer the following order: Page 0 section first, data section next and then the program section, though it is easier to follow the code if the program section precedes the data section as in the previous example.

The program section should start with the main code, i.e. the code you want to execute. The main code should be followed by various functions. The order is not important.

What is a function? A function, also known as a *subroutine* is a *self contained* code that implements a well defined functionality. What do I mean by self contained? You should be able to draw a line above and below your code for the function, and make sure that

1. Only way to branch **out of** the two lines is with `JSR` or `BSR` or `RTS` instructions. If you find any other branching instruction such as `BRA`, `BEQ` etc. then your code is most likely incorrect

2. Only way to branch **into** an instruction between the two lines from an instruction outside the two lines is with a `JSR` or `BSR` instruction. If you find any other branching instruction such as `BRA`, `BEQ` etc. then your code is most likely incorrect.

The function should terminate with a return from subroutine `RTS` instruction. It is a good programming practice not to have more than one `RTS` statement in any function.

Once you have written your function, it is there for you to use as many times as you need. To use the function, you should know where its first instruction is located in memory (technically known as the entry point). If you use the assembler to create the function, you can place a label before the very first instruction. The assembler will automatically `EQU`ate the label with the first instruction. If the function needs any additional information, they will have to be supplied by the user prior to using the function (technically known as binding). The function you will be writing will use one of the registers for binding. Also, many functions will return some useful value to the caller. In this case, it is a good idea to return the value in one of the registers.

Unless you write functions that do absolutely nothing (technically known as stubs), the function will use and modify one or more registers. If the caller happens to keep valuable data in one of these registers, then you have a potential problem. There are two possible solutions: The caller could save the values in the registers before calling your function, and then restore it after your function returns. Alternatively, your function can save the values in the registers it uses and then restore the values before it returns. The first approach more efficient but the second approach will result in fewer bugs. I strongly recommend that you get into the habit of writing functions that clean up after themselves and restore the registers the way they were before the functions used them[1].

[1]The only exception is the register that is used to return a value to the user. Clearly, these registers should not be restored to their original value!

Here are the basic rules for writing functions

1. Decide on its functionality. Don't try to create a Swiss army knife that has multiple functionalities built in. Your function must do only one thing, and it must do it well. Your documentation for the function must clearly state the functionality

2. Decide on its name. Pick a meaningful name but keep the name to 8 characters or less

3. Decide on the registers that will be used to pass information **to** the function. 8-bit values can be sent using **A** or **B** registers. 16-bit values can be sent using **X** or **Y** registers[2].

4. Decide what registers will be used to **return** values back to the caller. 8-bit values can be returned using **A** or **B** registers. 16-bit values can be returned using **X** or **Y** registers.

5. Decide what registers the function will use and which of these will be restored back at the end. Clearly document which registers will be used and **not** restored back as the registers that are modified by the function.

6. Write the function. Avoid the temptation of writing the function first and then worrying about the other items!

As an example, we will write a simple function called `RAND` that will return a random value every time it is called. The function remembers the last value it returned (this value can be initialized to a definite value such as zero) and uses it to generate the next value. The formula[3] it uses is simple: It shifts the last value left, and adds with carry the value 20. The function needs one byte of storage to keep track of the random value. This storage will be allocated in the data section using the `FCB` directive as

```
LASTRND     FCB     0
```

You use FCB to initialize consecutive memory loication (when you transfer the code from the PC). The label associated with the instruction is needed to determine the address where the instruction forms the constant. The assembler will automatically `EQU`ate the label with the address associated with the `FCB` directive.

[2]For example, `OUTA` expects the data in the**A** register, while `OUT1BYT` expects the information in the **X** register.

[3]I use this as a quick and dirty 8 bit random number generator. It is not the best, but the code is only 5 line long!

Type the following code, assemble it, transfer the S19 file to the HC11, and test your program by `CALL $2100`. Repeat[4] the `CALL` statement and verify that the value in the **A** changes after each call.

```
;;;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;;
    ORG $2100
    JSR RAND
    SWI


;;;;;;;;;;;;;;;;;;;;;;;;Start: RAND ;;;;;;;;;;;;;;;;;
; Function: RAND
; Purpose: Generate a random number
;
; Inputs: None
; Outputs: A random value returned in A registers
;
; Registers modified: A register (which has a random value)
;
; Memory usage: The most recently generated random number is
;                stored in memory with label LSTRAND
;                This value is used to generate the next value
;
; Notes: Not the best random number generator around but does a
;        halfway decent job.
;
;        works as follows:
;        shift the last random value left and add 20 with carry
;
;



RAND        LDAA    LASTRND
        LSLA
        ADCA    #20
        STAA    LASTRND ;don't forget to save it back
        RTS
;;;;;;;;;;;;;;;;;;;;;;;;End: RAND ;;;;;;;;;;;;;;;;;;;
```

[4] In BUFFALO, if you press the enter key at the prompt, BUFFALO will repeat the last instruction. So you don't have to type the CALL every time. Just press the enter key.

```
;
; DATA SECTION
;

    ORG $3000
LASTRND     FCB     0

;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;
```

3.5 Your second function

For your second function, you will be writing a function that will print an 8-bit value in HEX first and then in binary. We will call this function `PRBINARY`

An 8-bit number requires 2 hex-digits to print and BUFFALO has two routines to help you, one to print the left digit and the other to print the right digit. These are called `OUTLHLF` for out-left-half and `OUTRHLF` for out-right-half respectively.

To print it in binary, we will use the shift left instruction `LSL` . This instruction will shift all bits left by 1 place and the (left most) bit that is shifted out will be stored in the carry flag. I.e. if an 8-bit value before shifting was `abcdefgh` then the value after shifting will be `bcdefgh0`. Here each of the letters `a` to `h` represent bits. The carry flag will be set to `a`. Thus the value to be printed will be shifted left 8 times. After each shift, the A register will be loaded with either the ascii code for `0` or the code for `1` depending on whether the carry is cleared or set[5]. Note the function involves a counting loop.

We now have to decide register usage: We will use the **A** register to pass the value to the function. Internally, this value will be moved to **B** as **A** is needed in all the calls to BUFFALO routines. We need a counter, and we will use the **X** register to keep count. As a good programming practice, we will store and restore all registers we will use.

Here is the complete code for the function.

```
;;;;;;;;;;;;;;;;;;;;;Start: PRBINARY;;;;;;;;;;;;;;;;;;;
; Function: PRBINARY
; Purpose: To print a value in binary
;
; Inputs: Value to be printed is passed in the A register
```

[5]Conveniently, the code for `1` is one more than the code for `0`. So we load **A** with the code for `0` and add the carry to it

```
;
; Returns: None
;
; Registers affected: None. The values in the registers are stored
;             first and these values are restored at the end.
;
; Notes: The output consists of two parts. The value in A is first
;        printed in HEX, and then in binary
;

PRBINARY

; first save the registers we will be using: a, b and x
    PSHA
    PSHB
    PSHX

; copy a to be for later use

       TAB

; print a as hex number (2 digits).

; print the left digit
    JSR OUTLHLF

; the function destroys the value in a,
; so re load it! then print second digit
    TBA
    JSR OUTRHLF

; now print a colon and some spaces
    LDAA #':'
    JSR OUTA

    LDAA #' '
    JSR OUTA
    JSR OUTA
    JSR OUTA

; now print it in binary
```

```
; b has the value to be printed (recall the old tab)
;
; shift b  to the left by one bit and print '0' or '1' depending
; on what is in the carry flag
; repeat 8 times.
;
; we will use X register as counter
; to print what is in carry flag, we will load A
; with the code for '0' ; and add the carry to the code
; prior to calling OUTA

    LDX #8 *COUNTER

PRBLOOP CPX #0
    BEQ PRBDONE

    LDAA #'0'
    LSLB
    ADCA #0
    JSR OUTA
    DEX
    BRA PRBLOOP
PRBDONE
    JSR OUTCRLF

; restore the registers
    PULX
    PULB
    PULA
    RTS

;;;;;;;;;;;;;;;;;;;;;;;End: PRBINARY;;;;;;;;;;;;;;;;;;;
```

3.5.1 Test your function

We can test the function by loading different values in the A register. The random number generator we wrote first comes in useful here! Write the following program, and test it by repeating the call to $2100 from the BUFFALO prompt.

```
Code Deleted
```

3.6 Setting bits

We will now write a function that will set a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will set bit #4 in memory location $00. Recall that the bits are numbered right to left starting with bit #0. To set a bit, we use the `ORA` instruction. Write the following program, and test it by repeating the call to $2100 from the BUFFALO prompt.

```
Code Deleted
```

3.7 Clearing bits

We will now write a function that will clear a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will clear bit #4 in memory location $00. To clear a bit, we use the `AND` instruction. Write the following program, and test it by repeating the call to $2100 from the BUFFALO prompt.

```
Code Deleted
```

3.8 Toggling bits

We will now write a function that will toggle (make it zero if it was a one; make it one if it was a zero) a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will toggle bit #4 in memory location $00. To toggle a bit, we use the `eor` instruction. Write the following program, and test it by repeating the call to $2100 from the BUFFALO prompt.

```
Code Deleted
```

3.9 Testing bits

Often we have to take a decision based on whether a bit is set or not in memory. To test if one or more bits are set, we clear all other bits and see if the result is

zero. If so, none of these bits are set. If not, at least one of them was set. The following program will print `YES` if bit #4 in memory location $00 is set. Or else it will print `NO`.

```
Code Deleted
```

3.10 What you should turn in

A listing file, that contains all your functions. A copy of your interaction with the HC11 (log file if you are using the simulator) with explanation of what each task does.

Chapter 4

Programs

4.1 Objective

To become familiar with writing programs. This lab also illustrates the use of assembler directives.

4.2 What you should do

You will have to turn in your `LST` files for the programs you write for this lab. Check with the TA for additional instructions.

4.3 Programs

When you write programs in a high level language such as C, C++, Java, the compiler facilitates modular program development using function. In an earlier lab, we discussed some basic rules for writing functions. Another facility that higher level languages provides is the use of variables. Key features of variables in high level languages are:

1. Variables generally have a name and you use the name to manipulate them.

2. Variables have a type. The compiler keeps track of the type and makes sure that the way a variable is used conforms to its type.

3. Compiler negotiates with the operating system to obtain adequate memory for the variable.

4. Compiler makes sure that a memory allocated to a variable is not accidentally also allocated to another variable

5. Compiler keeps track of the life and scope of the variable so that the variable is available only when it is in the scope of the instruction (the simplest case is the distinction between global and local variables.

In assembly code, you are pretty much on your own. First of all, variables are known by their address. Some variables require more than one location. In this case, you allocate consecutive memory locations for the variable, and the address of the variable is the first of these locations. It is your responsibility to make sure that (a) you set aside adequate memory for the variable, and (b) you do not assign the same location for more than one variable. The most common errors that programmers make is not keeping this in mind. For example, the programmer may set the address of a two byte variable as `$3120` and the address of another variable as `$3121`. **It is a bad idea for you to manually assign addresses to variables.** Let the assembler do it for you. However, if you have to manually assign an address to a variable, `EQU`ate a label to the variable as

```
TOTAL EQU $3800 *variable to keep track of the total
```

The above instruction defines a variable called `TOTAL` that is stored in locations starting from `$3800`. Note you have no idea how many bytes are needed for the variable and therein lies the potential bug!

4.3.1 How the assembler works

Inside the assembler is a variable called the *location counter*, also known as the `dot` in the unix community. When the assmbler starts, the location counter is initialized to zero. As the assembler reads your program, the location counter changes in response to your code. For example, the `ORG` command sets the location counter to the value after the `ORG` instruction. Thus, if you want to change the location counter to `$00EB`, you would write

```
    ORG $00EB
```

The assembler keeps a copy of the HC11 memory internally, and the location counter is used to address the memory. Here, briefly, is how the assembler works:

1. Read the instruction. If the instruction has a label, and it is not an `EQU` instruction, then equate the label to the location counter. If the instruction is an `EQU` instruction, the label is `EQU`ated to value after the mnemonic.

2. If the instruction is a valid HC11 instruction, generate the code for the instruction. The code is stored in consecutive memory locations starting from the address given by the location counter. The location counter is incremented by the amount of memory needed to store code.

3. If the instruction is a `RMB` instruction, then add the value after the mnemonic to the location counter.

4. If the instruction is a `FCB` command, then a sequence of comma-separated values following mnemonic are stored in consecutive locations starting from the address given in the location counter. The location counter is incremented by the amount of memory needed to store these bytes. `FCC` is a convenient way to specify a sequence of ASCII characters. Thus the following two statements are equivalent

   ```
   FCB 72, 69, 76, 76, 79
   FCC /HELLO/
   ```

   The `FDB` instruction is similar to `FCB`, except the values are interpreted as 16-bit numbers. *These initializations are done in your PC and then transferred to the HC11 via the* `S19` *file. If these memory locations in your HC11 are modified after the transfer, either by your program or by accident, then you have to reload your* `S19` *file! Also, DO NOT use these commands to initialize variables. Your program will work only once. If you rerun your program, the variables will not be reinitialized!*

Here is an example of defining variables and constants (variables that your program will not modify).

```
        ORG $3000 ;Start of data section.
V1      RMB 4 ; set aside 4 bytes. EQUate V1 to first address
V2      RMB 11; set aside 11 bytes
THOU    FCB 3, $E8 ; Initialize 16-BIT variable called THOU
BUFF    FILL $22,18 ; same as fcb with $22 repeated 18 times
BUF2    RMB 20
        OPT s ; turn on symbol dump option
```

It is a good idea to turn on the symbol dump option. This will cause assembler to print all the symbols at the end of your program listing. Type in the above sequence of instructions and assemble it. Look at the `LST` file and write down what the symbols `V1`, `V2` etc., are `EQU`ated to and explain the results.

4.4 Your first program

We will now write the first program. When writing assembly code, a convenient way to document your code is to write the *pseudocode*. Rather than invent another

pseudo language, I will use a C like syntax[1]. The program we want to write should be similar to the following C program (since HC11 is essentially an 8-bit micro, all variables will be unsigned chars to keep things simple).

```
#include <stdio.h>
unsigned char v1, v2, v3, v4; unsigned char
total; main() {
    v1 = 11;
    v2 = 0x2F; /* In C prefix 0x denotes HEX */
    v3 = 'A';
    v4 = 044; /* In C prefix 0 denotes octal */
    total = v1+v2+v3+v4;
    printf("%02X", (unsigned) total);
    return 0;
}
```

Compile and run the above program using any C compiler. The output of your program should be `9F`. Now assemble the following HC11 program, and run it to verify that you get the same answer:

```
Code Deleted
```

Modify the C program so that `v3 = 'A';` is replaced by `v3=getchar();`. Run the C program and type the upper case letter `A` and press enter. You should see the same output as before. Now replace the instruction `ldaa #'A'` by `jsr inchar` and make sure that you equate `inchar` to `$ffcd`. Run the assembly above assembly language program and verify that it behaves the same as the C program.

4.5 On your own!

1. Write an assembly program that does the same as the following C program:

```
#include <stdio.h>

char v;
char total;
main() {
    v  = 0;
    total = 0;
```

[1] This reverses history! C language was invented to avoid writing assembly code!

```
v = v+1;
total = total + v;


v = v+1;
total = total + v;

v = v+1;
total = total + v;

v = v+1;
total = total + v;

v = v+1;
total = total + v;

printf("%02X", (unsigned) total);
return 0;
}
```

2. Modify the program so that the initialization `v = 0` is replaced by a call to `getchar()`. Run your C program and try different inputs at the keyboard. Make corresponding changes to the assembly language program and verify your results.

3. Modify the assembly language program so that you use a counting loop to loop 5 times over the basic code.

4. Modify the previous version (using a loop) where the number of times around the loop is in an 8-bit *variable* called `count`. Your program should initialize the variable to `5` so that the loop is executed 5 times.

5. A convenient way to get a value between 0 and 9 from the user is to use the instruction sequence

```
jsr inchar
anda #$0F
; in C, use: getchar() & 0X0F
```

   Use the above sequence to let the user specify the count. When expecting the user to enter a value, it is a good idea to prompt the user. Your code may look something like:...

```
Code Deleted
```

6. Write a program that will do the following:

   (a) Get a number between 0 and 9 from the user and store in a variable called `v1`.

   (b) Multiply `v1` by `4` and store the result in a variable called `v2`.

   (c) Add `v1` and `v2` and store the result in a variable called `v3`.

   (d) Multiply `v3` by `2` and store the result in a variable called `v4`.

   (e) Print the values in the variables separated by a comma. In the last lab, you saw an example for printing a colon. You can use the same approach.

   (f) Modify the program so that the output reads some thing like:

   ```
   v1 = 07
   v2 = 1C
   v3 = 23
   v4 = 46
   ```

7. Write a function that will be passed a value in **A** register. The function should return 10 times the value passed to it. Use the above function to write a program that does the following: The program should get a digit from the user, multiply it by 10 and store the value in a variable called `tens`. It should then get a digit from the user and add it to the variable `tens` and store it in a variable called `DecimalIn`. The program should then print the value in the variable.

8. Use the code you wrote to create a function that will read a two digit decimal number from the user. Call this function `ReadDecimal`. Write a program that will call this function to read a two digit number and store the value in a variable called `DecimalIn`. The program should then print the value in the variable.

9. Type the following C program, run it to see what the output is. Rewrite the program in assembly language and run it on the HC11. Verify that output of the assembly program matches the C program.

   ```
   #include <stdio.h>

   char delta, value;
   ```

```
char count;
main() {
    value = 0;
    delta = 1;
    count = 11;

foo:
    if (count == 0) goto bar;

    printf("%02X", (unsigned) value);
    putchar(','); putchar(' ');

    value = value + delta;
    delta = delta + 2;
    count = count -1;
    goto foo;

bar:
    return 0;

}
```

10. Type the following C program, run it to see what the output is. Rewrite the program in assembly language and run it on the HC11. Verify that output of the assembly program matches the C program.

```
/*

   I decided to use fprintf(stderr instead of printf(
   as printf( and getche don't mix well.

   I am forced to use getche instead of getchar because
   getchar    puts the terminal in the 'cooked' mode rather
   than 'raw' mode    and there seems to be no way to uncook
   the input in MS Windows.

   The program as written will work on all variants of Windows
   and on unix boxes if you know the right curses!

   Note: getche is an exact equivalent of INCHAR in BUFFALO
         To print a string, you use OUTSTRG in BUFFALO
```

```
*/

#include <stdio.h>
#include <conio.h>

char c;

main() {
foo:
    fprintf(stderr,"\n\nWelcome! Your choices:\n\n");
    fprintf(stderr,"\n1. Set temperature");
    fprintf(stderr,"\n2. Set Speed\n"
    fprintf(stderr,"\nChoice please: ");
    c = getche();

    if (c == '1') goto one;
    if (c == '2') goto two;
    fprintf(stderr,"\n\nNot a valid choice!\n");
    goto foo;

one:
    fprintf(stderr,"Good choice. \n");
    goto more;
two:
        fprintf(stderr,"Try later\n");


more:
    fprintf(stderr,"Try again? ");
    c = getche();
    if (c == 'y') goto foo;
    if (c == 'Y') goto foo;
    if (c == 'n') goto bye;
    if (c == 'N') goto bye;
    goto more;

bye:
    return;
}
```

Chapter 5

Tables

5.1 Objective

To become familiar with table driven code.

5.2 What you should do

You will have to turn in your `LST` files for the programs you write for this lab. Check with the TA for additional instructions. In most of the examples, the name is left blank as `____________`. Make sure you enter your name in its place.

5.3 Tables

What is a table? Technically, table is same as an array except we use the term *table* to refer to arrays whose elements are constants. To operate on the table, we need two pieces of information, *where, in memory, the table starts(* **the starting address***)* , and *how many elements are in the table (* **the size or dimension***)*. In this lab, we will initially consider the case where each element requires one byte of memory.

5.4 Setting up a table

To setup a table in memory, we generally use `FCB`, `FDB` and `FCC`. For example, if we want to setup a table of prime numbers, we would write (using decimal since it is easier to read):

. . .

```
    ORG $3000 *data section
...
...
primes fcb 2,3,5, 7, 11, 13, 17, 19, 23, 29 *table of some primes
nprimes equ 10 *number of entries in the table
```

If you want to set up a table of ascii code for digits, you would write

```
...
    ORG $3000 *data section
...
...
digits1 fcb  $30, $31, $32, $33, $34, $35, $36, $37, $38, $39
ndigits1 equ 10 *number of entries in the table
```

A **better** way to do the same is to write

```
...
    ORG $3000 *data section ..$
...
...
digits fcc  /0123456789/
ndigits equ 10 *number of entries in the table
```

Assembler will convert `FCC` to a sequence of `FCB`.

Exercise: Write an ASM file with the two versions of the digits tables, assemble the file and look at the `LST` file. Verify that the two tables are identical.

5.5 Working with tables

We will write functions to perform some basic tasks with the tables. In all these functions, we will use the following convention:

1. The starting address will be passed to the function in the **X** register.
2. The size of the table will be passed to the function in the **B** register.

5.5.1 Table lookup

This is the simplest and the most useful function. We want to know if an element is in the table. For now, we will work with a table of 8-bit quantities. The value to be looked up will be passed in the **A** register. The function will have to return

a Yes/No value. A convenient way to return a Yes/No value is to use a hardware flag. Let us use the **C**arry flag. The function will set the flag if the answer is yes; or else it will clear the carry.

```
Code Deleted
```

Here is a program that uses the function:

```
Code Deleted
```

Exercise: Type the above program, assemble it, transfer the S19 file to the 68HC11, and run the program with CALL 2100. When the program starts, type the following text Pack my box with five dozen liquor jugs.

Exercise: The lookup program affects the **X** and **B** registers. Modify the function so that the function initially stores these two registers in the stack, and restores them before returning. Verify that the program works correctly.

Exercise: Modify the program by adding another table, the table of symmetric characters: AHIMOTUVWXYimnouvwxy. The program should, in addition to checking to see if a character is a vowel, it should also check to see if it is a symmetric character. Run the program and enter the following text: Axiomboard. You should get an output similar to the following:

```
 done
 >c 2100

 =====================================
Lab on using Tables

Name: _____________________
This program is an infinite loop!
Hit the reset button to quit
=====================================


A  is a vowel and is a symmetric character
x  is not a vowel and is a symmetric character
i  is a vowel and is a symmetric character
o  is a vowel and is a symmetric character
```

```
m  is not a vowel and is a symmetric character
b  is not a vowel and is not a  symmetric character
o  is a vowel and is a symmetric character
a  is a vowel and is not a  symmetric character
r  is not a vowel and is not a  symmetric character
d  is not a vowel and is not a  symmetric character
```

5.5.2 Input with validation

Another use of the table lookup is to validate input from the user. Suppose we want the user to enter a social security number. In this case, we want to make sure that we accept only digits and ignore non-digits (the user may be in the habit of typing spaces, dashes etc. You want to silently ignore these). So it will be useful to write a function, called `rddigit` that will accept only digits. The following program shows a typical usage:

```
;Name:
;email:
;date:
;
; Standard buffalo equates
; Make sure you have ALL the equates in the file.
;

ucase       equ $ffa0
wchek       equ $ffa3
dchek       equ $ffa6
init        equ $ffa9
input       equ $ffac
output      equ $ffaf
outlhlf     equ $ffb2
outrhlf     equ $ffb5
outa        equ $ffb8
out1byt     equ $ffbb
out1bsp     equ $ffbe
out2bsp     equ $ffc1
outcrlf     equ $ffc4
outstrg     equ $ffc7
outstrgo    equ $ffca
inchar      equ $ffcd
vecinit     equ $ffd0
```

```
    org $3000
; setup some strings ...
preamble
    fcc /=====================================/
    fcb 10
    fcc /Lab on using Tables/
    fcb 10, 10 *use 10 to start a new line
    fcc /Name: _____________________/
    fcb 10
    fcc /This program is an infinite loop! /
    fcb 10
    fcc /Hit the reset button to quit/
    fcb 10
    fcc /=====================================/
    fcb 10,10,10, 4


; setup the table of digits

digits  fcc /0123456789/
ndigits equ 10


    org $2100
    ldx #preamble
    jsr outstrg
mainloop
    jsr rddigit
    bra mainloop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; rddigit: Behaves like inchar, except ignores non-digits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rddigit jsr input *This does not wait for the user
    tsta
    beq rddigit *Looks like the user has not typed anything
;
; if we get here, the user typed something. Verify it first
;
```

```
    ldx #digits
    ldab #ndigits
    jsr lookup
    bcc rddigit *oops, not in the table. Go back for more
;
; if we get here, the input was ok
; echo it back as the user would like some feedback
;

    jsr     outa
    rts


;;; Add the code for lookup function here
```

Exercise: Type the above program and run it. Enter the following input and explain what you see:

Exercise: Type the above program and run it. Enter the following input and explain what you see: `123-34-1879`.

Exercise: Modify the above program so that it accepts exactly 10 digits and stops with an `SWI` after reading 10 characters. In other words, convert the main loop into a counting loop. *You should keep the count in the* **B** *register.* Verify your program with the input: `823--xx-34-1879`.

5.5.3 Translations using tables

Some time we want to translate a value to another value. For example if the user types the character `8`, your program will receive the ascii code for the character. You now will have to convert it to its value, viz `8`. This is easy since you just have to subtract `$30`. However, if the user enters data in HEX, then he would expect the program to translate `A` and `a` to 10, `B` and `b` to 11 and so on. For problems of this nature, we first write down the translation table

value	translation
$30 or '0'	0
$31 or '1'	1
$32 or '2'	2
etc.	etc
$39 or '9'	9
$41 or 'A'	10
$61 or 'a'	10
$42 or 'B'	11
$62 or 'b'	11
etc.	etc
etc.	etc

In our assembly program we set up two tables. It is extremely important that the two tables be ordered as follows: The table of values first immediately followed by the table of translations. For example, to perform the above translation, we will write

```
Code Deleted
```

To illustrate the use of the translation table, let us write a function that will translate telephone numbers. For example, if the input to the program is a mixture of numbers and letters as in `1-800-CALLATT` the program should translate it to `1-800-2255288`. The translation table can be found on any telephone and is:

value	translation
'A', 'B', 'C'	2
'D', 'E', 'F'	3
'G', 'H', 'I'	4
'J', 'K', 'L'	5
'M', 'N', 'O'	6
'P', 'Q', 'R', 'S'	7
'T', 'U', 'V'	8
'W', 'X', 'Y', 'Z'	9

The following program sets up the above table and uses it to translate phone numbers.

```
Code Deleted
```

Exercise: Type the above program and enter the input: `1-800-UMD-ALUM`. You should see the following output. Clearly explain how the translation gets done.

```
 =====================================
Lab on using Tables

Name: _____________________
This program is an infinite loop!
Hit the reset button to quit
=====================================



 1  1
 -  -
 8  8
 0  0
 0  0
 -  -
 U  8
 M  6
 D  3
 -  -
 A  2
 L  5
 U  8
 M  6
```

Chapter 6

Parallel Input/Output

6.1 Objective

To become familiar with hardware interfacing and input/output. Also introduces polling timer overflow flag.

6.2 Getting started

It is extremely important that you become familiar with your HC11. Since most of you will be using the `CMD11E1` board made by Axiom Manufacturing, this write-up is specific to the board.

1. Locate the jumper `JP13` and make sure that it is open.

2. Locate the MCU port. This is a dual row 17-pin Berg style connector. Locate pin #1 on the port. This is identified with the number 1 on the front of the board. On the other side (solder side), pin #1 is identified by a square solder.

3. Connect a ribbon cable to the port. Use a continuity tester to locate and identify the following pins: pin 1 (PA0), pin 5 (PA4), pin 6 (PA5), pins 9 and 10 (5 Volts), pins 11 and 12 (Ground). (See page 15 of the User's guide that came with your board.)

4. Make two test light emitting diode probes. It is a good idea to have several probes handy. To make a test probe, connect a 3.3K resistor to the **anode** of a light emitting diode (See figure 6.1). Use a light emitting diode with an operating voltage of 1.7 volts and a current rating in the 1-5 mA range. If you use an light emitting diode with 20+ mA operating current, the light

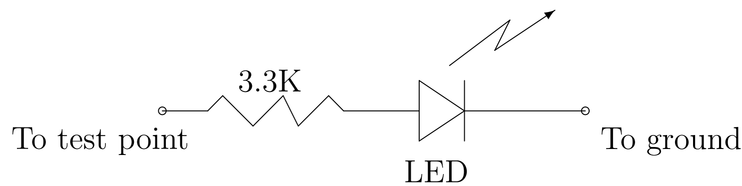

Figure 6.1: *Test Probe. Make sure that the resistor is connected to the anode of the light emitting diode. The longer lead of the light emitting diode is the anode. To test a pin, connect the cathode to ground and the resistor to the pin.*

emitting diode would be dim when it lights up and you may have to look carefully to see if it is on. Test the probe by connecting the free end of the resistor to pin 9 and the free end of the light emitting diode to pin 11. The light emitting diode should light-up. If not, the chances are you have connected the resistor to the cathode of the light emitting diode. Redo your circuit by connecting the resistor to the anode.

6.3 Experiment

6.4 Hardware connection

1. **Disconnect the power to HC11.**
2. Connect a (LEFT) test probe between pin 5 and pin 11. The cathode of the light emitting diode should be connected to pin 11 and the free end of the resistor to pin 5.
3. Connect a (RIGHT) test probe between pin 6 and pin 12. The cathode of the light emitting diode should be connected to pin 12 and the free end of the resistor to pin 6.
4. Label the two light emitting diodes as LEFT and RIGHT so that it is easy to identify them (you can use a magic tape and small pieces of paper).
5. Connect the input circuit as shown in figure 6.2.
6. Check and recheck all your connections before you connect the power to the HC11.

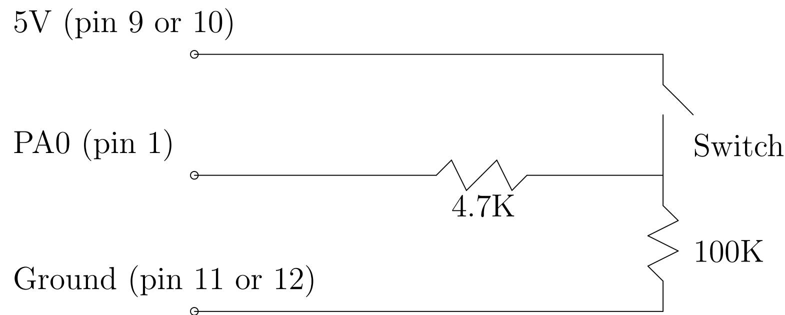

Figure 6.2: *Typical input connection*

6.5 Theory of operation

Port A in HC11 appears at memory location `$1000`. In port A, the bits 0,1 and 2 (referred to as PA0, PA1 and PA2) are inputs. They appear as pins 1, 2, and 3 of the MCU port (this is specific to the Axiom Board. If you use a different board, check the documentation that came with your board). Thus if you read the memory location `$1000`, then the value in the bits depends on the voltage applied to the corresponding pins. If the voltage is 5 Volts, the bit will be one, and if the voltage is zero, the bit would be zero.

Bits 3,4,5 and 6 of port A (referred to as PA3, PA4, PA5 and PA6) are outputs. This means your program can control the voltage in the corresponding pins 4,5,6 and 7 by writing a 1 or a zero to the appropriate bit in location `$1000`.

6.6 Experiment 1

Use memory modify, `MM $1000`, command to modify the location `$1000`. Change the value in the location to `$00`, `$10`, `$20`, and `$30`. (Note: When you communicate directly with the HC11 using BUFFALO commands, you do not type the `$`.) After each change, look at the state of the light emitting diodes and write down what you see. Provide a brief explanation of what you see.

Using the methods you learnt in earlier labs, write a program that will read the keyboard and depending on what the user types, perform the following:

Key	Action
`Q` or `q`	Turn on the LEFT led. The state of other led should not change.
`Z` or `Z`	Turn off the LEFT led. The state of other led should not change.
`E` or `e`	Turn on the RIGHT led. The state of other led should not change.
`C` or `c`	Turn off the RIGHT led. The state of other led should not change.

6.7 Experiment 2

Use memory dump to see the contents of `$1000`. Close the input switch and dump the contents of location `$1000`. Repeat the experiment with the switch open. Write down what you see and provide a brief explanation of your observation.

Write a program that will, in an infinite loop, print either `CLOSED` or `OPEN`, depending on the state of the switch. The program should continually monitor the state of the switch, and print `CLOSED` is the switch is closed. Or else it should print `OPEN`.

6.8 Timing

In this part of the lab, we will monitor the `TOF` flag. This flag is controlled by the *free running counter*. The free running counter is a 16 bit counter that counts the clock ticks. It is set to `$0000` on power up. It counts up to `$FFFF`and then rolls over to `$0000` and the process is repeated until you power off the HC11. Every time the counter rolls over, it sets the `TOF` flag. This flag is not automatically reset and it is your responsibility to reset it in your code depending on your need. To reset the flag you have to write a 1 to it [1].

Your HC11 most likely uses a 8 MHz crystal which means that the processor speed is 2 MHz, or you get 2×10^6 clock ticks every second. The free running counter rolls over every 2^{16} ticks, or you get

$$\frac{2 \times 10^6}{2^{16}} = 30.52 \text{ overflows every second}$$

We can use this to create a $\frac{30.52}{2}$ Hz square wave by toggling an output pin every time counter rolls over. Here is the code (you have to fill in the details!)

```
Code Deleted
```

Type and run the above program. Connect the LED to PA4 (pin #5) and you should see it flicker. Connect the pin to a oscilloscope and verify that you are generating a square wave. Verify that the frequency is correct.

[1] This is true of all HC11 flags. You reset a flag, i.e. make it go to zero, by writing a 1 to it!

Chapter 7

Interrupt Processing

7.1 Objective

To become familiar with interrupt processing.

7.2 Background

In an earlier lab, you had to generate a 30.52 Hz square wave signal on PA4 pin. The code for generating the square wave is given below for your reference. Make sure that you run the program and verify that you get the square wave before proceeding further. Also, to understand this lab, you must connect the PA4 pin to a oscilloscope and see the square waves.

```
Code Deleted
```

7.3 Interrupts

If you study the above code, most of the time is spent waiting for the clock to rollover (the loop at `LOOP2`). This is a lot like sitting in front of the clock and watching and waiting for the clock to rollover. Or, for that matter, sitting in front of a stove and watching and waiting for kettle to boil, or watching food being cooked in a microwave oven. A better solution would be to go about ones job and arrange matters so that one is told when an event occurs (the clock chimes, the kettle whistles, the microwave oven sounds an alarm etc.). When we are told that the event has occurred, we then take appropriate action (turn off the stove, take food out of the oven etc.). Most of the HC11 interrupts work the same way. In essence this is what happens:

1. When an event occurs, a flag is set. For example, the clock rollover sets the `TOF` flag.

2. Associated with the flag is a masking bit. The name of the bit is the same as the flag except the final `F` is replaced by `I`. The mask associated with `TOF` is `TOI`.

   (a) If the mask is zero, then nothing much happens. The event is ignored by the interrupt processing structure.

   (b) However, if the mask is set, then request for service is generated.

      i. The `I` bit in the CCR is a master disable switch. If this is set (by using the command `SEI`), then the request for service does not interrupt the computer and is hence ignored.

      ii. However if `I` bit is cleared (by using the command `CLI`), then the CPU is interrupted.

   Note: It is extremely important that you have an `SEI` before any code that can turn on an interrupt and and `CLI` after all relevant and required initialization is performed.

3. When a CPU is interrupted, it stops its current task and starts the service.

4. When performing the service, you get a completely new set of registers. So you can not assume that the registers will have any specific value. Also, when the service terminates, the new set of registers is destroyed. So you can not assume that the rest of the code can see what you stored in the register as part of the service. In fact, the interrupted task would be oblivious to the fact that a service was provided. The only way it can find out is if the service modifies some memory location.

5. The location of the service that is associated with a particular interrupt is defined by the hardware manufacturer, and is called the *jump vector*. This would be in read only memory and can not be chanted. The operating system, BUFFALO, sets the start of the service to a known location and sets aside three bytes at that location. Your service will start with `JMP` instruction to the actual code for the service. Your service **must end with the `RTI` instruction**.

   The address of services to three important interrupts is given in the following table:

Interrupt	**Service location**
TOF	$00D0
RTIF	$00EB
OC2F	$00DC

6. Your service code should, as part of the service, *turn off* the flag that generated the interrupt. If not, the request for service will still be active and will generate a new service request as soon as the current service end!

Here is a short checklist for what you should do:

1. In your main routine, enable an interrupt by turning on the associated mask.
2. Write the service routine. As part of the service make sure you turn off the flag that generated the interrupt.
3. Let HC11 know where to find the service. In other words, Link the service to the request.

With this background, we will modify the square wave generator to use the interrupt. Here is the complete code with some of the standard equates left out (you need to have them at the top of the file!). You should compare this code with the earlier one. In this code, the main program does nothing really interesting. It just prints a series of `Z`'s to the screen

```
Code Deleted
```

How does this code differ from the previous one? We don't wait for the clock to rollover. Instead, the main program goes about its task (in this case not a very interesting one!). Note that although the service routine uses the **A** register, this does not affect the value the main routine has stored in the **A** register (the character `Z`).

7.4 The Real time interrupt

The real time interrupt, `RTI` (not to be confused with the `RTI` instruction) acts exactly like the timer overflow interrupt, except you can control the time between interrupts using the last two bits (0 and 1) of `PACTL` at location `$1026`. If the both the bits are set, the time between the interrupts is is 32.768 ms, i.e. same as timer overflow. However, you can decrease the time between the interrupts (increase the rate) by changing the last two bits as `PACTL` as shown below:

Last two bits of PACTL	**Time between interrupts**
00	4.096 ms (244.1 Hz)
01	8.192 ms (122 Hz)
10	16.384 ms (61 Hz)
11	32.768 ms (30.5 Hz)

Thus if we want to use the `RTI` interrupt, we have to change the above code as shown below. Note the crucial differences:

```
Code Deleted
```

7.4.1 Exercises

1. Modify the above program to generate a 61 Hz square wave by setting the RTI interrupt rate to 61 Hz.

2. After you made the modification, toggle PA4 every 61 interrupts (Hint, set up a counter and initialize it to 61 in the main program. In the interrupt service, decrement the counter. When the counter reaches zero, toggle the pin and reset the counter back to 61). Verify that the signal you generate is a 1 Hz square wave

3. Create a simple clock. In addition to toggling the pin, increment an 8-bit variable called `TIME`. In the main loop, instead of printing `Z`'s, print the variable using `OUT1BSP`.

7.5 The output compare interrupt

The HC11 has 5 `OCx` interrupts. These are like alarm clocks. You set a desired 'alarm' time and when the clock matches the alarm setting, the `OCxF` flag will be turned on and could then generate a request for service. Note that if you do not change the alarm setting, you will still get an interrupt every 32.768 ms. However, having the alarm gives you greater flexibility. First a code that does not change the alarm setting and hence generates 30.5 Hz square wave.

```
Code Deleted
```

Verify that the above code also generates a 30.5 Hz square wave. Now we can reset the alarm to get a different frequency. For example, if we modify the service routine as follows, we will get an interrupt every 2000 clock ticks or every 1 ms for a 1 K Hz signal.

```
Code Deleted
```

Chapter 8

Signal Generation

8.1 Objective

To become familiar with generating square waves.

8.2 Background

In an earlier lab, you had to generate a square wave signal using various interrupts. This lab builds on this. For a general square wave signal, we define the on-time, T_{on}, the off-time, T_{off}, and the period T as shown in the figure 8.1. The goal of the lab is to generate such square waves. The ratio $\frac{T_{\text{on}}}{T}$ is called the `duty cycle` and is expressed as a percentage. When dealing with time it is convenient to talk in terms of clock ticks. The HC11 has a 2 MHz e-clock, or you have 2×10^6 ticks per second. Hence 1 clock tick is 0.5 microseconds.

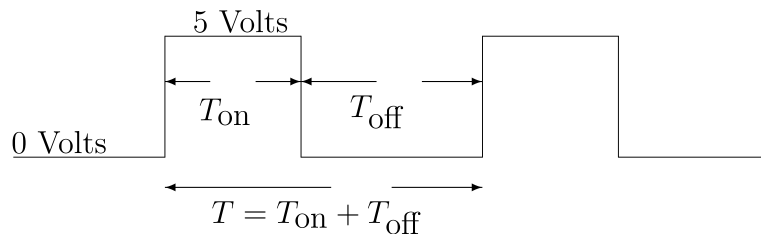

Figure 8.1: 0-5 volt square wave

8.3 Variable frequency signal generator

In this experiment, we will generate a square wave with frequency selected by the user. We will fix the duty cycle at 25%. To get started, we will first write the code for generating a single tone.

8.4 500 Hz tone generator

Here is a typical calculation to generate a 500 Hz signal:

$$
\begin{aligned}
T &= 1/500 = 2 \times 10^{-3} \text{ seconds} = (2 \times 10^{-3}) \times (2 \times 10^{6}) = 4000 \text{ ticks} \\
T_{\text{on}} &= 4000/4 = 1000 \text{ ticks} \\
T_{\text{off}} &= 4000 - 1000 = 3000 \text{ ticks}
\end{aligned}
$$

To generate a 50 Hz, 25% duty cycle signal we use two variables called `ONTIME` and `OFFTIME`. Every time we get an `OC2` interrupt, we toggle the pin `PA4` as before. We check to see if we turned the pin `ON` or `OFF`. If we turned it on, then we schedule the next interrupt to occur `ONTIME` ticks later. However, If we turned it off, then we schedule the next interrupt to occur `OFFTIME` ticks later. Thus, the earlier code that was used to generate a square wave is modified as follows:

```
Code Deleted
```

Assemble and run the above program. Connect `PA4` to an oscilloscope and verify that the duty cycle and the frequency are correct.

8.5 Variable frequency generator

We now modify the above code to create a variable frequency generator. The program will monitor the keyboard and depending on the number the user enters, it will change the frequency as shown in the table below:

Number	Frequency	Period seconds	Period ticks	On time ticks	Off time ticks
0	440	0.002273	4545	1136	3409
1	466	0.002145	4290	1073	3217
2	494	0.002025	4050	1013	3037
3	523	0.001911	3822	956	2866
4	554	0.001804	3608	902	2706
5	587	0.001703	3405	851	2554
6	622	0.001607	3214	804	2410
7	659	0.001517	3034	759	2275
8	698	0.001432	2863	716	2147
9	740	0.001351	2703	676	2027

As a programmer we are only interested in the on-time and off-time. We use `FDB` to create two tables in the data section as shown below:

```
ONTIMETBL
    FDB 1136 ,1073, 1013, 956, 902
    FDB 851, 804, 759, 716, 676
OFFTIMETBL
    FDB 3409, 3217, 3037, 2866, 2706
    FDB 2554, 2410, 2275, 2147, 2027
```

Note that it makes sense to enter numbers in decimal notation. Each entry in the table requires two bytes (we use 16 bit numbers to measure time since the HC11 clock is a 16 bit quantity). Hence we use `FDB` instead of `FCB`. Note that this also means that we have to index through memory in steps of **two bytes**. Suppose we want to access the element #4 in ONTIMETBL and the value #4 is in the **B** register (the number #4 is used only as an illustration. In the application the register **B** will have the value). Then we have to access the element we have to write

```
LDX #ONTIMETBL
ABX
ABX ; NEED TWO ABX'S
??? 0,X
```

We can now modify the single tone generator to a programmable square wave generator by making the following changes:

Before		After

LDD #1000
STD ONTIME
LDD #3000
STD OFFTIME
		LDX #ONTIMETBL
		LDY #OFFTIMETBL
		LDD 0,X
		STD ONTIME
		LDD 0,Y
		STD OFFTIME

	Before			After
	LDAA #'Z'		LOOP	
LOOP	JSR OUTA			JSR INPUT
	BRA LOOP			TSTA
				BEQ LOOP
				ANDA #$0F
				TAB ; B HAS INDEX
				LDX #ONTIMETBL
				ABX
				ABX
				LDY #OFFTIMETBL
				ABY
				ABY
				LDD 0,X
				STD ONTIME
				LDD 0,Y
				STD OFFTIME
				BRA LOOP

8.6 Exercises

1. Make the changes shown above and run the program. Connect `PA4` to an oscilloscope. Press any of the keys `0` to `9` and verify that the frequency changes.

2. Change the program so that when you press any of the keys `0` to `9`, the frequency should be fixed at 100 Hz but the duty cycle changes. Pick 10

different duty cycles. Also, connect the output pin to a digital voltmeter. How does the voltage change with the duty cycle?