Topic 2 –Microcontroller Programming

Required supporting materials

- This Module and any supplementary material provided by the instructor
- Device documentation provided in the appendix of this CoursePack
- CNT MC9S12XDP512 Development Kit and 12 VDC Power Adapter
- USBDM Pod or BDM Pod and “A to B” USB Cable
- CodeWarrior

Rationale

Well-structured and documented code results in dependable and maintainable systems.

Expected Outcomes

The following course outcome will be addressed by this module:

Outcome #1: Develop and debug assembly language programs using an Integrated Development Environment (IDE).

Outcome #2: Create assembly language programs that manipulate data using operations and expressions.

Outcome #3: Interface with onboard, simple GPIO, and programmable devices.

As this course progresses, you will refine the basic skills and understanding of embedded systems through programming the 9S12X using S12XCPU Assembly Language and ANSI C.

Connection Activity

You have now learned enough about the 9S12X and its Assembly Language to create simple I/O tasks. More complex tasks may require careful pre-planning, more instructions, a clearer understanding of the ways in which address locations are accessed, and a better understanding of the ways in which program flow can be controlled. The more complex the software, the more careful you will need to be in structuring and documenting it. You will discover that certain tasks are used repetitively or have the potential to be used in different software routines – these should be stored in such a way that they can be accessed without needing to copy or re-enter the code. A well-structured program should be easily understood, easily operated, and easily maintained.

Assembly Language Fundamentals

When programming in S12XCPU Assembly Language there are two fundamental types of commands: Assembler Directives and Instructions.

Assembler Directives

Assembler directives are commands that control the development software on our computer, called the Assembler. Assembler Directives do not end up in the code that the microprocessor runs. Some of the more common Assembler Directives are:

- INCLUDE –tells the assembler to add the contents of an external file
- EQU –assigns a label to a particular address
- ORG –tells the assembler to move to an address location before continuing
- DS.B nn –defines storage space for nn Bytes (8 bits), and should be in RAM
- DS.W nn –defines storage space for nn Words (16 bits)
- DS.L nn –defines storage space for nn Longs (32 bits)
- DC.B *val(s)* –defines a constant Byte or Bytes (8 bits) and should be in ROM
- DC.W *val(s)* –defines a constant Word or Words (16 bits)
- DC.L *val(s)* –defines a constant Long or Longs (32 bits)

There are a lot of other Assembler Directives, which can be found in the "S12(X) Assembler Manual" from Freescale. Here's the link:

http://cache.freescale.com/files/soft_dev_tools/doc/ref_manual/CW_Assembler_HC12_RM.pdf

There's also a link to this 400 page document in Moodle. It should also be available in the lab in case you need it.

One useful feature of the S12(X) Assembler is its ability to do math on the fly. You can get it to calculate addresses or offsets while it is creating the machine code for the CPU, which can make your life a bit easier.

Instructions

Instructions, unlike assembler directives, are translated into machine language for the microprocessor to carry out.

A summary of the S12XCPU Assembly Language Instruction Set can be found in Appendix A of the Reference Manual, the link to which you've been given already. Let's look at what we can learn about a particular instruction from this guide.

To understand the instruction set, we need to look at the explanatory notes that precede it, on pages 2 – 5 of the Guide. You'll get to do that in the exercise that follows.

You also need to know a bit more about the terminology used in Assembly Language programming.

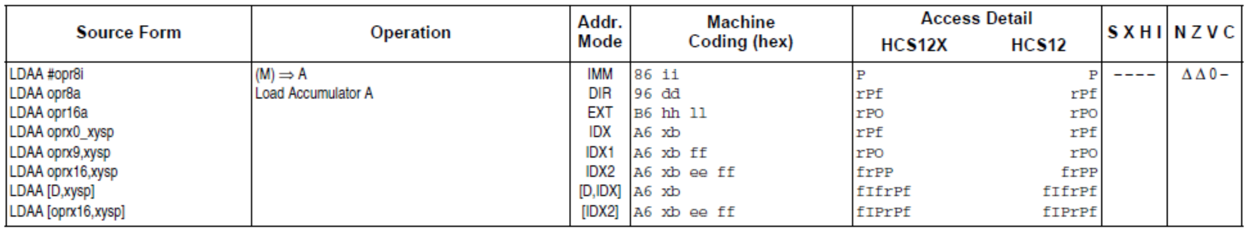
Op Code – Short for Operation Code, this refers to something the microprocessor will interpret as an instruction. Each version of each instruction will have a unique op code, which determines what else the microprocessor needs to look at in order to carry out the instruction.

Post Byte – Some op codes tell the microprocessor to read the next byte to get details on the operation to be carried out. In the Instruction Set, this will be indicated in the "Machine Coding" column as "eb", "lb", or "xb", depending on the type of post byte.

Mnemonic – the Assembly Language Mnemonic is the pseudo-English abbreviation that represents a particular op code or set of related op codes and their post-bytes. Programming in Assembly Language involves getting to know the Mnemonics and figuring out what each of the variants for that instruction does and what it needs.

Operand – Some, but not all, instructions require something to work on. This could be actual data, it could be an address containing the necessary data, or it could be an offset from some other point of reference.

Here's an example from the Reference Guide: LDAA. Some pertinent points follow.



Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
LDAA #opr8i	$(M) \Rightarrow A$	IMM	86 ii	P	P	----	ΔΔ0–
LDAA opr8a	Load Accumulator A	DIR	96 dd	rPf	rPf		
LDAA opr16a		EXT	B6 hh ll	rPO	rPO		
LDAA oprx0_xysp		IDX	A6 xb	rPf	rPf		
LDAA oprx9,xysp		IDX1	A6 xb ff	rPO	rPO		
LDAA oprx16,xysp		IDX2	A6 xb ee ff	frPP	frPP		
LDAA [D,xysp]		[D,IDX]	A6 xb	fIfrPf	fIfrPf		
LDAA [oprx16,xysp]		[IDX2]	A6 xb ee ff	fIPrPf	fIPrPf		

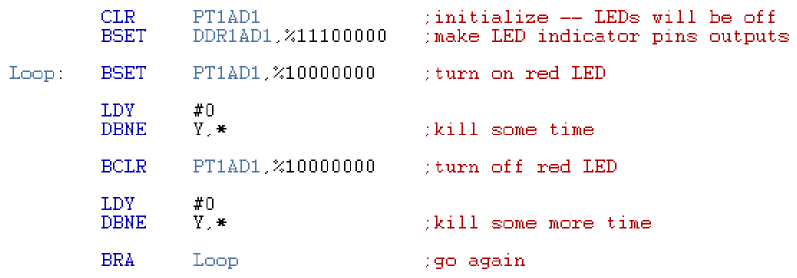
- There are eight different ways that "LDAA" can be interpreted by the Assembler. You can think of these as "overloads" in .net terminology.
- The first and simplest of these uses the "IMM" addressing mode. This means that the accumulator will be loaded with the contents of the address directly following the instruction. From the first column, you will notice that this requires a "#" sign in front of the next byte. Since A is an 8-bit register, it can only load 8-bit data.
- The source form "#opr8i" tells you that the instruction is made of a single-byte op-code followed by an 8-bit immediate value for an operand.
- "86 ii" shows that the actual op code is $86, and "ii" means two nibbles (a byte) immediately following the op code.
- The "EXT" mode is used to access the 8-bit value contained at a particular 16-bit address. "opr16a" means that this version of the command has a single-byte op-code followed by a 16-bit address. "B6 hh ll" tells you that this version of the command has the op-code $B6, and that the address will be two high nibbles followed by two low nibbles. This, by the way, indicates that this is a Motorola-type device, and uses "big-endian" address formats as opposed to Intel-type devices, which use "little-endian" addresses, read low-byte first followed by the high-byte.
- The "Access Detail" column tells you how many bus clock cycles this command takes, one cycle per letter code, and what happens for each clock cycle (something we usually don't need to know much about). Remember that the bus clock is half of the crystal frequency. Since the crystal on our board is 16 MHz, the bus clock is 8 MHz, with a period of 125 ns. So, the "IDX2" version of this command would take four clock cycles ("frPP") at 125 ns per cycle for a total of 500 ns.
- The last two columns tell us what to expect in the Condition Code Register. In this case, we should expect to see changes for the "negative" and "zero" flags, and the "overflow" flag will always be cleared to zero.

The following should be a review of work done in a previous course, but is included here as a reminder as to how to get started in CodeWarrior.

1. Start a project in CodeWarrior. (Typical settings shown in italics: Select the right microcontroller – *MC9S12XDP512*; select the right connection pod – *TBDML*; select the right core configuration – *Single Core*; select the right language – *Absolute assembly*; enter an appropriate project name and location – *Desktop–>9S12X–>Projects*.)
2. Skeleton.txt is a file that should be available in Moodle. Open and copy its contents (*Ctrl A – Ctrl C*); replace the text in main.asm with these contents (*Ctrl A – Ctrl V*).

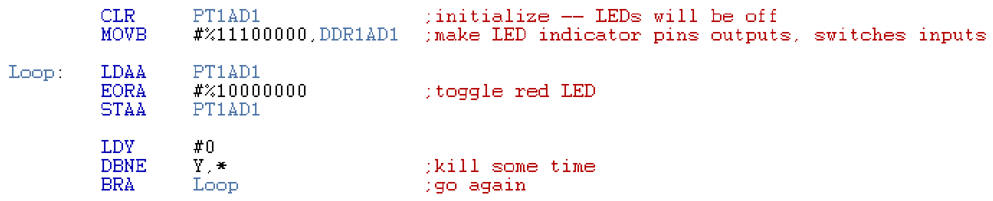
3. Change the information in the file header to reflect who you are and the nature of the project.
4. Enter your code in ROM, which will be where the skeleton file says "Main:".

Here's a bit of code you can put into a project to practice with some of the concepts covered to this point. You might want to determine how long "kill some time" takes, based on what you now know about the timing of clock cycles and the size of the *Y* register. (You should come up with about a 25 ms delay.)



```
        CLR     PT1AD1              ;initialize -- LEDs will be off
        BSET    DDR1AD1,%11100000   ;make LED indicator pins outputs

Loop:   BSET    PT1AD1,%10000000    ;turn on red LED

        LDY     #0
        DBNE    Y,*                 ;kill some time

        BCLR    PT1AD1,%10000000    ;turn off red LED

        LDY     #0
        DBNE    Y,*                 ;kill some more time

        BRA     Loop                ;go again
```

Here's some code that performs exactly the same function as the code above. See if you can explain why.

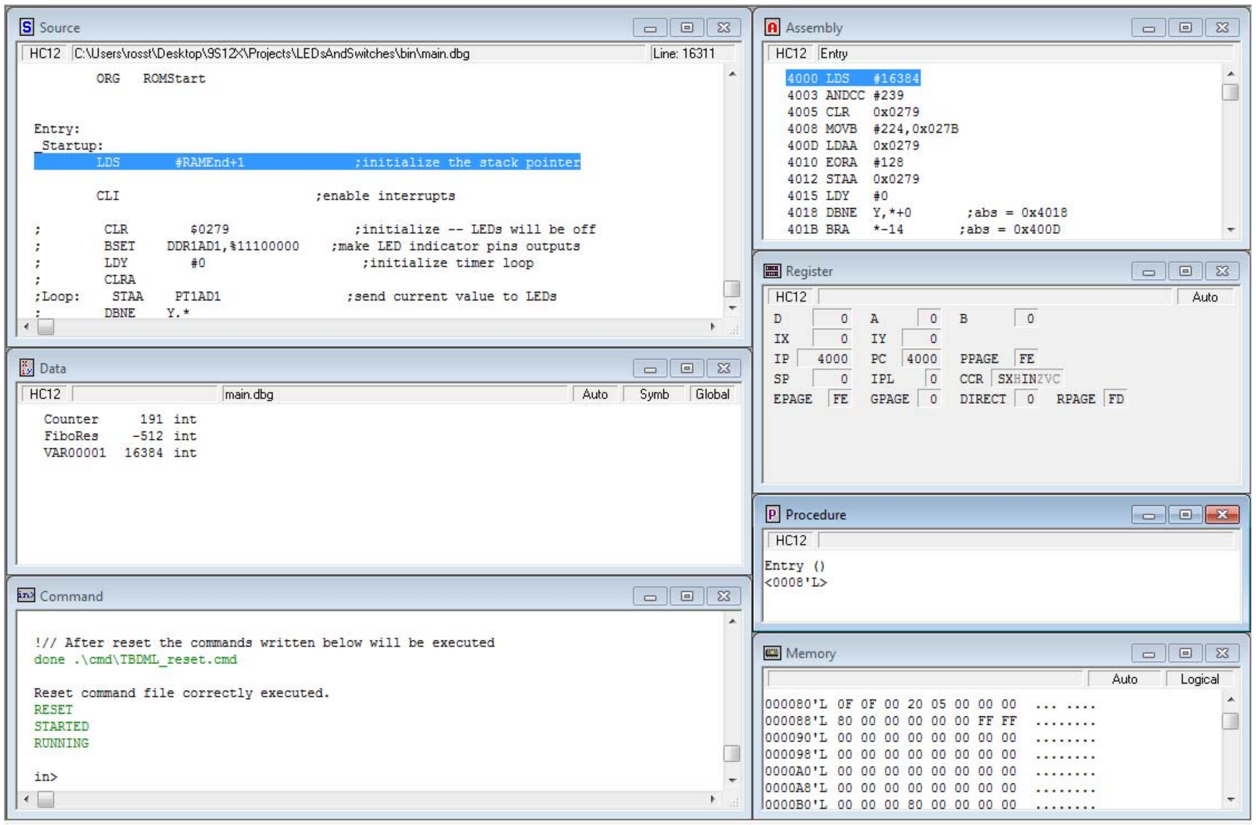


```
        CLR     PT1AD1              ;initialize -- LEDs will be off
        MOVB    #%11100000,DDR1AD1  ;make LED indicator pins outputs, switches inputs

Loop:   LDAA    PT1AD1
        EORA    #%10000000          ;toggle red LED
        STAA    PT1AD1

        LDY     #0
        DBNE    Y,*                 ;kill some time
        BRA     Loop                ;go again
```

Rudimentary Debugging Skills

In the Debug window, across the top tool bar, you will find the following set of buttons. You can mouse-click these or use the associated hot-keys, shown in the following table.

Button	Function	Hot Key
➡	Run	F5
⤵	Single Step	F11
⇉	Step Over (run subroutine)	F10
⤴	Step Out (exit subroutine)	Shift+F11
⤵	Assembly Step	Ctrl+F11
⊣	Halt	F6
➲	Reset	Ctrl+R


Source

HC12 C:\Users\rosst\Desktop\9S12X\Projects\LEDsAndSwitches\bin\main.dbg Line: 16311

```
        ORG   ROMStart


Entry:
 Startup:
        LDS       #RAMEnd+1             ;initialize the stack pointer

        CLI                          ;enable interrupts

;        CLR       $0279               ;initialize -- LEDs will be off
;        BSET    DDR1AD1,%11100000    ;make LED indicator pins outputs
;        LDY       #0                   ;initialize timer loop
;        CLRA
;Loop:    STAA    PT1AD1                ;send current value to LEDs
;        DBNE    Y.*
```

Assembly

HC12 Entry

```
4000 LDS   #16384
4003 ANDCC #239
4005 CLR   0x0279
4008 MOVB  #224,0x027B
400D LDAA  0x0279
4010 EORA  #128
4012 STAA  0x0279
4015 LDY   #0
4018 DBNE  Y,*+0       ;abs = 0x4018
401B BRA   *-14       ;abs = 0x400D
```

Register

HC12 Auto

```
D        0   A        0   B        0
IX       0   IY       0
IP    4000   PC    4000   PPAGE   FE
SP       0   IPL      0   CCR    SXHIN2VC
EPAGE   FE   GPAGE    0   DIRECT   0   RPAGE  FD
```

Data

HC12 main.dbg Auto Symb Global

```
Counter     191  int
FiboRes    -512  int
VAR00001  16384  int
```

Procedure

HC12

```
Entry ()
<0008'L>
```

Command

```
!// After reset the commands written below will be executed
done .\cmd\TBDML_reset.cmd

Reset command file correctly executed.
RESET
STARTED
RUNNING

in>
```

Memory

Auto Logical

```
000080'L 0F 0F 00 20 05 00 00 00  ... ....
000088'L 80 00 00 00 00 00 FF FF  ........
000090'L 00 00 00 00 00 00 00 00  ........
000098'L 00 00 00 00 00 00 00 00  ........
0000A0'L 00 00 00 00 00 00 00 00  ........
0000A8'L 00 00 00 00 00 00 00 00  ........
0000B0'L 00 00 00 80 00 00 00 00  ........
```

- In the Debug environment, the "Source" and "Assembly" windows show you the code as typed by you and as interpreted by the Assembler.
- The "Data" window shows you the contents of variables and constants used in the program. These are updated whenever the program is halted or reaches a breakpoint.
- The "Memory" window shows the contents of any memory location, and highlights recent changes in red.
- The "Register" window shows the contents of all of the microprocessor's registers. A very good troubleshooting technique is to check the contents of the registers against what you think you're loading into them. This will help you determine if you've made

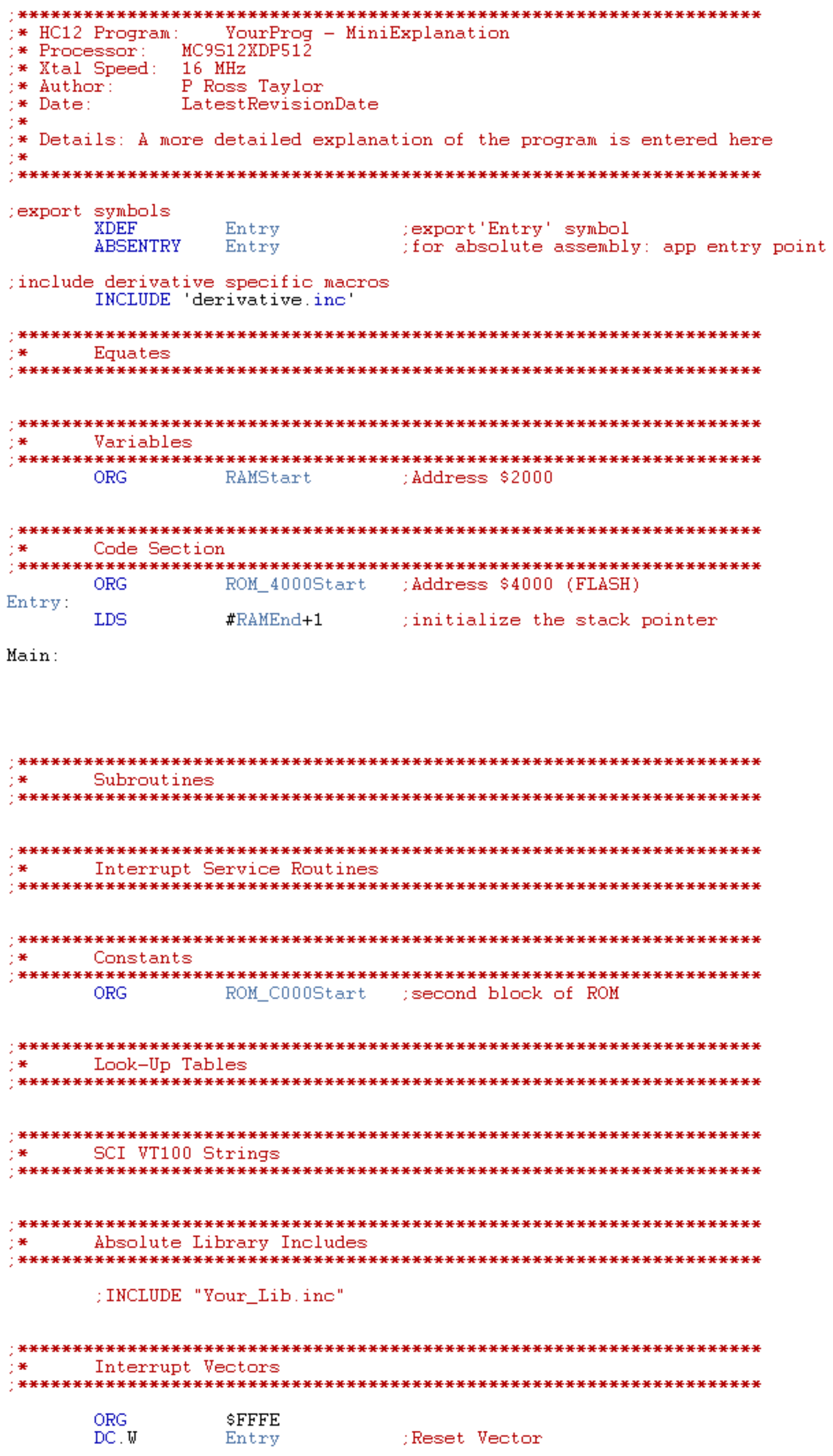
an error in loading something using Immediate addressing mode (#) or Extended addressing mode (contents of a memory location indicated by the address).

- In the “Source” window, you can right-click on a given line and set a breakpoint as a temporary stopping point in the program, allowing you to examine the contents of the registers, data, and memory.
- While the microprocessor’s activity is halted, you can manually change the contents of the registers and memory, which will allow you to do “what if” scenarios or cut down the number of cycles in a long loop by changing the value of a register that’s being used as a counter.

Once you compile and download code to your microcontroller, it will continue to run that code on start-up until you over-write it. You’ve burned your program into EEPROM on board, and, until you reprogram it, it will continue to run the same instructions faithfully.

Documentation and Comments

In a previous course, you used a “skeleton” file each time you started a new project. The following is a skeleton file modified from one used by Marc Anderson at NAIT. This is a good starting point – change the text and tabs, etc. to match your comfort zone, and save this as a simple .txt file for future use. It may also be provided by your instructor.



```
;********************************************************************
;* HC12 Program:    YourProg - MiniExplanation
;* Processor:   MC9S12XDP512
;* Xtal Speed:  16 MHz
;* Author:      P Ross Taylor
;* Date:        LatestRevisionDate
;*
;* Details: A more detailed explanation of the program is entered here
;*
;********************************************************************

;export symbols
        XDEF        Entry           ;export'Entry' symbol
        ABSENTRY    Entry           ;for absolute assembly: app entry point

;include derivative specific macros
        INCLUDE 'derivative.inc'

;********************************************************************
;*      Equates
;********************************************************************


;********************************************************************
;*      Variables
;********************************************************************
        ORG         RAMStart        ;Address $2000


;********************************************************************
;*      Code Section
;********************************************************************
        ORG         ROM_4000Start   ;Address $4000 (FLASH)
Entry:
        LDS         #RAMEnd+1       ;initialize the stack pointer

Main:




;********************************************************************
;*      Subroutines
;********************************************************************


;********************************************************************
;*      Interrupt Service Routines
;********************************************************************


;********************************************************************
;*      Constants
;********************************************************************
        ORG         ROM_C000Start   ;second block of ROM


;********************************************************************
;*      Look-Up Tables
;********************************************************************


;********************************************************************
;*      SCI VT100 Strings
;********************************************************************


;********************************************************************
;*      Absolute Library Includes
;********************************************************************

        ;INCLUDE "Your_Lib.inc"


;********************************************************************
;*      Interrupt Vectors
;********************************************************************

        ORG         $FFFE
        DC.W        Entry           ;Reset Vector
```

Using the Skeleton.txt File

1. Start a project in CodeWarrior following steps you've previously used. (Typical settings shown in italics: Select the right microcontroller – *MC9S12XDP512*; select the right connection pod – *TBDML*; select the right core configuration – *Single Core*; select the right language – *Absolute assembly*; enter an appropriate project name and location – *Desktop–>9S12X–>Projects*.)
2. Open and copy the contents of Skeleton.txt (*Ctrl A – Ctrl C*); replace the text in main.asm with the copied contents (*Ctrl A – Ctrl V*).
3. Change the information in the file header to reflect who you are and the nature of the project.
4. To include a library, first right-click the "Includes" in the Browser and locate the file (*should be in Desktop–>9S12X–>Libraries*), then in main.asm insert an INCLUDE Assembler directive following the commented template shown.
5. Enter your code in ROM, which will be where the skeleton file says "Main:".
6. Declare any variables in RAM where the skeleton file indicates "Variables" using a "DS.x nn" Assembler directive.
7. Define any constants in ROM where the skeleton file indicates "Constants" or "Look-up Tables" or "SCI VT100 Strings" using "DC.x" and the actual constant data.
8. Put any locally-defined subroutines after the end of your main code loop, which will automatically happen if you use the skeleton area labelled "Subroutines".

When you write a subroutine, you should write a header that tells a programmer how to use the routine and what to expect of it. The following is an example.

```
;*****************************************************************
;*                 HexToBCD                                      *
;*                                                               *
;*Regs affected:  D (A and B)                                    *
;*                                                               *
;*A hexadecimal value arrives in Accumulator D and is converted   *
;*to a 16-bit BCD, returned in D                                 *
;*                                                               *
;*Maximum hexadecimal value allowed is $270F                      *
;*                                                               *
;*****************************************************************
```

From this, the programmer knows that the D Accumulator must be loaded with the appropriate hexadecimal 16-bit word, and that, after a "JSR HexToBCD" the D Accumulator will contain the 16-bit BCD equivalent. You should also know, as a programmer, that since the contents of D are modified, the A and B registers will be modified by this subroutine.

You will be writing some subroutines that are specific to the task at hand – these will go in the "main.asm" file you're working on, usually close to the end of the code. You will also be writing subroutines that can be used in multiple projects. These you will collect into "libraries" of subroutines, which you will link to the main file using assembler directives. When you write a subroutine, then, you should determine whether it could be used by other programs and should be in a library or if it is unlikely to be used elsewhere and should therefore just be locally-accessible.

As you write code, get in the habit of writing comments as you go. Make your comments informative, not just a rewording of the Assembly instructions. In the following example, the first line is not informative; the second one is.

```
Bad:  LDAA  #$E0        ;load accumulator A with hex E0

Good: LDAA  #%11100000  ;ready to initialize Port AD Data Direction Register
```

Don’t type pages of code and then try to go back and insert comments. The comments are there to help you keep track of what you’re doing as you are coding just as much as to help win your instructor’s favour!

Of course, if your Main code is set up as a carefully-planned sequence of calls to well-named subroutines (more on that later), comments would be redundant.

```
Start:      JSR   SwLED_Init
Flash:      JSR   Red_On
            JSR   1msDelay
            JSR   GrnLEDOn
            JSR   1msDelay
            JSR   All_Off
            JSR   1msDelay
            BRA   Flash
```

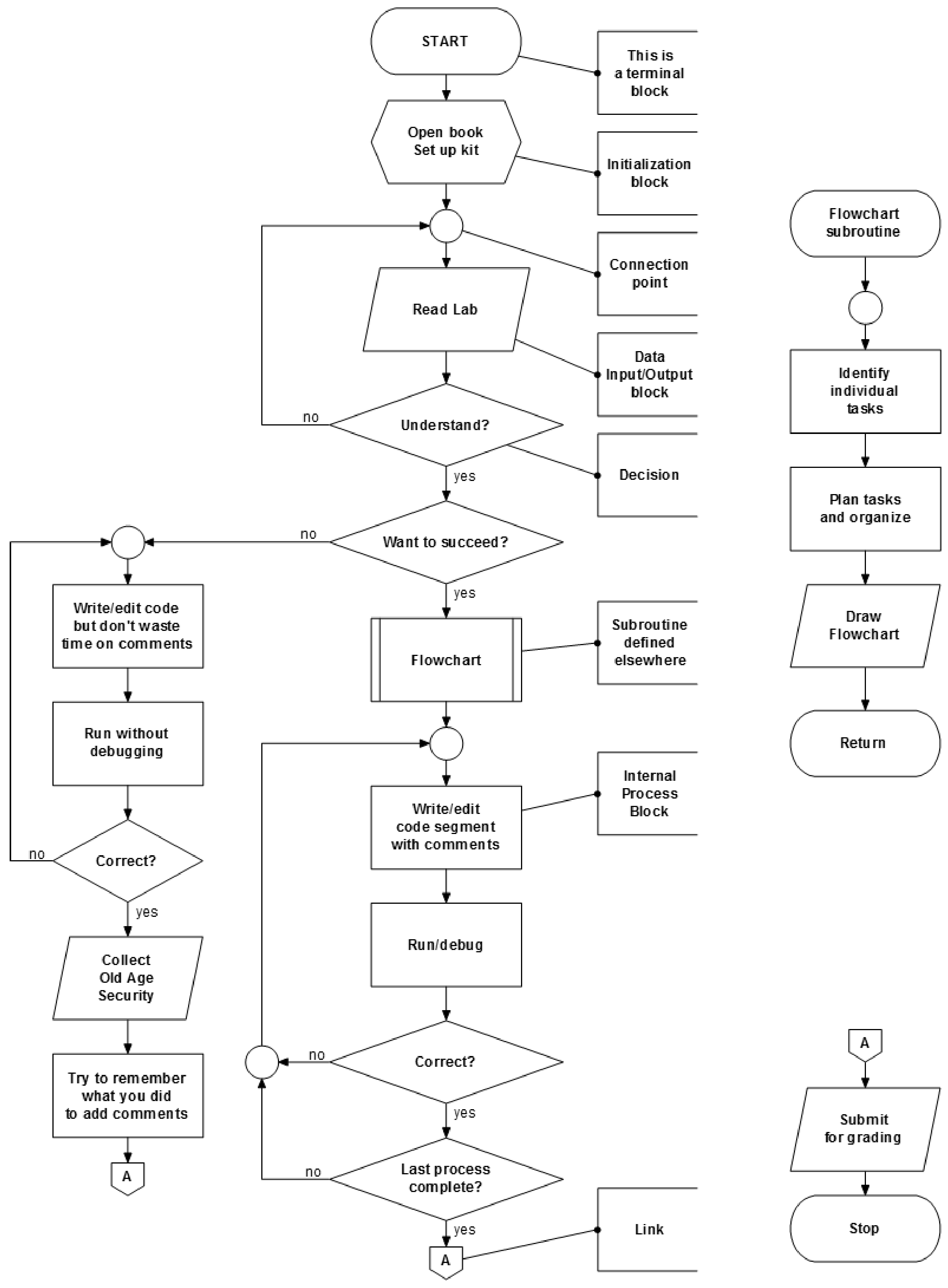
This kind of code wouldn’t need comments, as it is self-commenting.

As your programs become more involved, you will need to do some pre-planning, as you would with any programming task in any language.

Some programmers are comfortable with writing pseudo-code as a guide to eventually developing proper code. Others prefer using flow-charts. Either way, a properly planned program will have cleaner code, will be more likely to run without errors, will be easier to troubleshoot, and will be easier to modify if the specifications change.

The following page points out some of the pitfalls of programming without proper planning.

Flowcharting



The flowcharting sequence and the clearly-recognizable blocks shown on the previous page provide a good way for you to organize your thoughts and your code.

When you attack a programming problem, one of the first things you should do is identify discrete tasks that need to be completed. Now, look over your list of tasks: are there any that could be made into general subroutines for other kinds of projects? If so, these should be written as generically as possible, for inclusion in libraries.

Consider writing your program so that the Main program is, for the most part, a sequence of calls to subroutines. Draw your flowchart to reflect this flow of events. You can put all the "special" subroutines (i.e. the ones not in libraries) below the Main code (well-marked and documented, of course).

Don't put "code snippets" into your flowchart – this should be understandable to someone who is knowledgeable about programming but doesn't necessarily know the language you're using. Instead, put descriptive terms or phrases in the program. For example, don't say "Carry Flag Set?" Instead, say what that carry flag means in terms of the program. It may mean "Data Ready?" or "Counter Max'd out?", or whatever your program is looking for.

Subroutines

Often when programming a microcontroller, you encounter pieces of code that are used in multiple places. Rather than doing the "cut'n'paste" routine, which results in very long and unreadable code, you can write subroutines (think "methods" in C#) which can be called from the Main program anytime you wish.

In fact, well-structured code should have a very simple Main program that calls well-named subroutines to do all the work. We'll get into proper code structuring later, after spending time writing useful subroutines. The following general pointers should help you immensely.

- A subroutine needs a unique label – use something informative, like "CheckLeftSw:". In this context, labels are followed by a colon.
- You must enter a subroutine using a ***JSR*** (ok, you could also use ***BSR***, but it's not designed to jump more than 256 addresses from where you are, and takes no more effort or time than a JSR, so why would you bother?).
- You must exit a subroutine using ***RTS***.

  Important Note!!! You must not use any of the "branch on decision" instructions to enter or exit a subroutine. When you JSR into a subroutine, the microprocessor places the return address onto the stack. When you RTS from a subroutine, the microprocessor grabs the return address off of the stack and goes back to where it came from. If you don't have an RTS to match every JSR, you will mess up the stack. You will either add more and more stuff to the stack resulting in a stack overflow, or you will take too much stuff off the stack, straying into unknown territory in memory. Either way, your program will crash in microseconds.

  If you use an accumulator or register within the subroutine, ***PSH*** it onto the stack before you use it, then ***PUL*** it back off the stack when you're done with it so that it's back to the condition it was in before you entered the subroutine. (That is, unless you want to use that register to return a value from the subroutine.)

  Bottom line: make sure that you always unstack exactly the same number of items as you have stacked, and in reverse order.
- If you have subroutines within the file called Main, put them all below the actual Main program in a section clearly labelled "Subroutines".

- Make sure you have a descriptive header that explains what the subroutine does and what registers, if any, are modified by the subroutine. This way, when you decide to use the subroutine somewhere else, you'll know what it expects and what it returns.
- Where possible, try to make your subroutines, particularly ones used in libraries, broadly useful. Typically you wouldn't put commands into a subroutine that make it so it can only be used in one place in one program, unless it helps clarify the general operation of the program.
- If you need to change a subroutine, particularly one in a library, make sure the changes are backwards compatible so that previous programs that use these subroutines will still operate. If there is no way of keeping a subroutine backwards compatible, create a new subroutine with a different name for use in subsequent programs.
- In the context we've chosen for development (Absolute Assembly), you only have access to global variables. Where possible, try to make your subroutines work without using variables so that they are more portable. If you must use a variable or constant, make sure you put a note in the header reminding yourself or someone else using your subroutine that the variable or constant needs to be declared in the Main program.

Libraries of Subroutines

As previously mentioned, some subroutines should be made available so they can be used by other programs. These library files are simply text files containing the subroutines you want to include in them. The Assembler/Linker is designed to include any libraries you want to attach to your main code when you Run/Debug your program. Please note that everything in the library gets added into your assembled code, so you might want to plan your libraries so as to keep the amount of unused code in your assembled code to a reasonable amount.

As previously mentioned, you need to do the following two things to include a library file:

- Add the library to the "includes" folder in the Project window (right-click the folder and add the file when prompted).
- Put an "INCLUDE" Assembler directive ***after*** your code, so that the included subroutines will appear at the end of your code in ROM.

If you use the Skeleton file mentioned earlier, it has a pre-defined block for these INCLUDE statements.

One way to create a library is within the context of a Main program. Write all your subroutines below the Main program, as usual. After you have tested each of them and are convinced they do what you want them to do, move the subroutines into a separate text file and create an informative comment block at the top – it should list each of the subroutines contained in the library – and save the result as a ".inc" file. It's that easy!

Alternatively, you can start a new ".inc" file, and build it up alongside a "main.asm" file that checks each routine as you build it. You'll need to make the appropriate "includes" to make this work. This author recommends this method, because it reduces the number of surprises you might experience.

The more difficult part is deciding what you want to have in a library. You should collect together subroutines that are related, and are therefore likely to be used for a particular type of program. For example, you will be doing a lot of work with the Serial Communication Interface (SCI). It would make sense to have routines that send and receive characters through the SCI in the same library. It doesn't make sense to have routines that turn on the LEDs in that library.

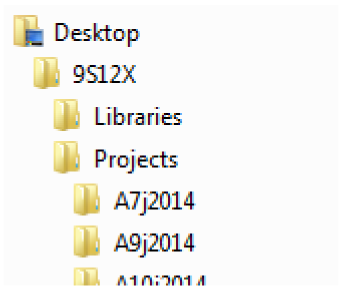
Assembly language libraries suited to the content of this course would include the following:

```
SwLED_Lib.inc
Misc_Lib.inc
SevSeg_Lib.inc
LCD_Lib.inc
SCI0_Lib.inc
PWM_Lib.inc
ATD0_Lib.inc
IIC0_Lib.inc
```

The focus of this course will shift to ANSI C programming before all of the material in this list of libraries has been covered, so you will likely not be required to create all of these libraries yourself. However, your instructor may provide you with these and other libraries as seems appropriate.

Misc_Lib.inc will contain routines that could be used in a number of types of programs – things like HexToAsc, HexToBCD, etc.

If you haven't already done so, you should develop a useful file structure for your work, like the following:



Create this file structure, and simply copy the entire thing back and forth between the desktop of the computer you're working on and your file storage device.

You should probably store your skeleton file in the "9S12X" directory so you can access it every time you start a project.

During the creation of a new project, the CodeWarrior IDE will create the folder for that project and will build all the associated subfolders and files inside the project folder.

S12XCPU Addressing Modes

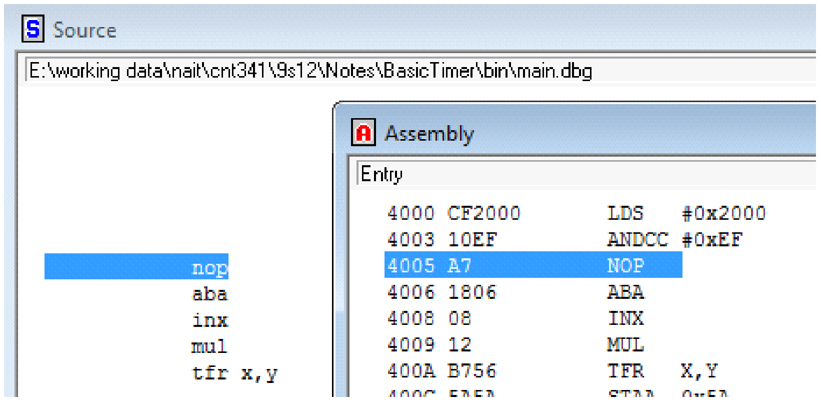
Addressing modes define what memory the instruction will operate on. Each instruction offers one or more addressing modes. The S12XCPU offers some very complex addressing modes, and we will not look at all of them. Some addressing modes lend themselves well to compiler output, so as humans, we aren't suitable candidates for their use.

The assembler selects the addressing mode, where appropriate, from the form of the source instruction. The text form of the instruction entered into the assembler is ultimately rendered into machine code – that which the CPU understands. Common omissions or 'trivial' mistakes in code entry can lead to incorrect values or incorrect addressing modes in machine code.

Inherent - INH

The simplest of all addressing modes is inherent (INH). Inherent instructions require no additional information to operate.

The following code snippet contains a number of Inherent commands. Notice how the Assembler interprets each – no reference to memory addresses.



Source
E:\working data\nait\cnt341\9s12\Notes\BasicTimer\bin\main.dbg
Assembly
Entry

```
nop
aba
inx
mul
tfr x,y
```

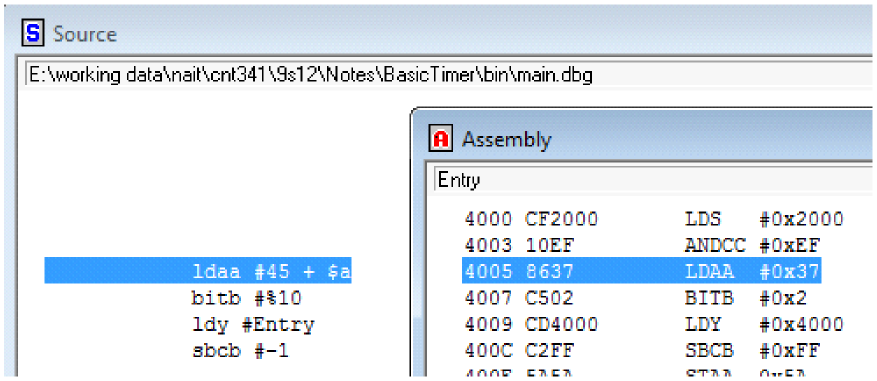
```
4000 CF2000      LDS   #0x2000
4003 10EF        ANDCC #0xEF
4005 A7          NOP
4006 1806        ABA
4008 08          INX
4009 12          MUL
400A B756        TFR   X,Y
```

Immediate - IMM

The immediate addressing mode contains the required operands in the object code, meaning the required information is constant, user-defined, and part of the instruction.

To indicate the immediate addressing mode, these instructions must use a pound sign on the operand. This will differentiate the immediate form from the extended form, which we will look at next.

The following code snippet contains a number of commands in Immediate Addressing mode. Notice how the Assembler interprets each command, and what it will work on.



Source
E:\working data\nait\cnt341\9s12\Notes\BasicTimer\bin\main.dbg
Assembly
Entry

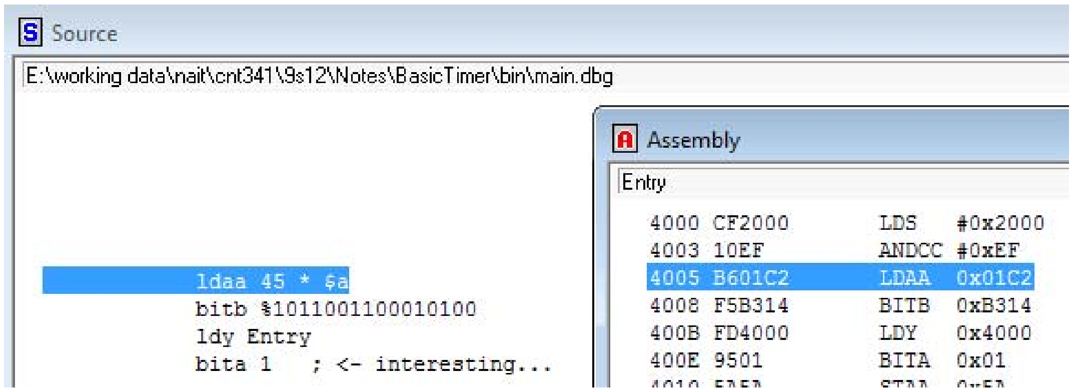
```
ldaa #45 + $a
bitb #%10
ldy #Entry
sbcb #-1
```

```
4000 CF2000      LDS   #0x2000
4003 10EF        ANDCC #0xEF
4005 8637        LDAA  #0x37
4007 C502        BITB  #0x2
4009 CD4000      LDY   #0x4000
400C C2FF        SBCB  #0xFF
```

Extended – EXT

The extended addressing mode requires a 16-bit address. The byte(s) at this address are used by the instruction. What happens to the byte(s) depends on the instruction.

The assembly language form for extended addressing requires no decoration, just anything that can be bent into an address:



E:\working data\nait\cnt341\9s12\Notes\BasicTimer\bin\main.dbg

```
ldaa 45 * $a
bitb %1011001100010100
ldy Entry
bita 1   ; <- interesting...
```

Entry

```
4000 CF2000     LDS   #0x2000
4003 10EF       ANDCC #0xEF
4005 B601C2     LDAA  0x01C2
4008 F5B314     BITB  0xB314
400B FD4000     LDY   0x4000
400E 9501       BITA  0x01
```

We can use labels to define constants (in ROM) and variables (in RAM), then we can use various addressing modes, like “EXT”, to access these. Here’s an example.

```
            ORG   $2000             ;start of RAM
Counter:    DS.B  1                 ;one byte assigned as variable Counter
            ORG   $4000             ;start of FLASH for program
            MOVB  #$5A,Counter      ;place initial value into Counter
```

Direct – DIR

The direct addressing mode is used to operate on memory locations 0x0000 – 0x00FF. Syntactically this form is identical to extended, except this form requires one less byte of code, as the high byte of the address is assumed to be $00. This addressing mode is useful when RAM is available in the first 256 bytes of the memory map, as it provides fast access for variables. Sadly, for the S12XCPU as configured in the 9S12XDP512, there is not a lot of call for direct addressing, since the “first page” of memory contains the microcontroller’s internal peripheral module registers. The assembler will automatically detect and implement direct addressing instead of extended if the instruction supports it.

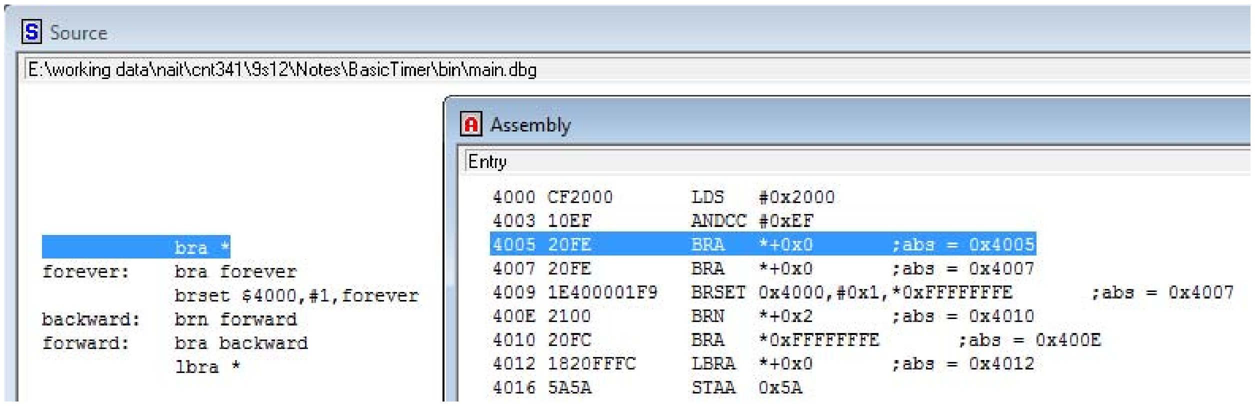
Relative – REL

The relative addressing mode is used principally in branching instructions. The S12XCPU supports long and short branching. The operand(s) in a branch instruction form a signed offset that participates in forming a new address for the program counter – in other words, the program execution moves to a new point, or branches away. The target address is found by adding the relative offset in the operand(s) to the address following the first (maybe only) offset operand. One of the things you should appreciate about the assembler is its ability to calculate relative offsets for you. You put in labels, it does the math.

Note: Some instructions like BRSET and BRCLR don’t show the addressing mode as REL because they are performing two commands: a compare and a branch. However, they are using relative addressing, so you won’t be able to use these commands to move more than −128 or +127 counts of the program counter from where you’re at. This can be a serious limitation, and will probably catch you off guard: you’ll have a program that’s working perfectly, then you'll add in a bit of code between one of these commands and its target. Suddenly, you'll get an "out of range" error, with no simple way of solving the problem.

The simple branching commands have a "long branch" option. By putting an "L" in front of the mnemonic, its range becomes –32,768 to +32,767. This isn't available for the complex instructions.

In the code snippet below, by comparing the "Source" window to the "Assembly" window, you can see how the Assembler has interpreted each of these instructions using Relative addressing. Take some time to get to know what the offsets are and how they are displayed in the disassembled code, in the comments, and in the machine code. Some of these are discussed below.


Source
E:\working data\nait\cnt341\9s12\Notes\BasicTimer\bin\main.dbg
Assembly
Entry

```
           bra *
forever:   bra forever
           brset $4000,#1,forever
backward:  brn forward
forward:   bra backward
           lbra *
```

```
4000 CF2000      LDS   #0x2000
4003 10EF        ANDCC #0xEF
4005 20FE        BRA   *+0x0        ;abs = 0x4005
4007 20FE        BRA   *+0x0        ;abs = 0x4007
4009 1E400001F9  BRSET 0x4000,#0x1,*0xFFFFFFFE       ;abs = 0x4007
400E 2100        BRN   *+0x2        ;abs = 0x4010
4010 20FC        BRA   *0xFFFFFFFE       ;abs = 0x400E
4012 1820FFFC    LBRA  *+0x0        ;abs = 0x4012
4016 5A5A        STAA  0x5A
```

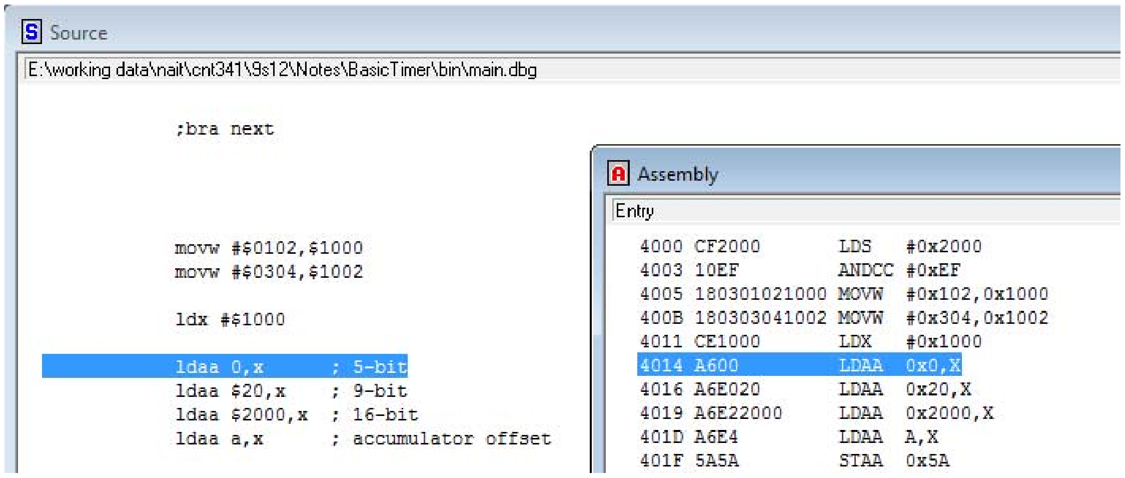
Look at the "**bra ***" line. As written, this means we want the code to branch back to the beginning of the current line. In the disassembled code, this shows up as "BRA *+0x0", which again means the intent is to branch 0 positions from the beginning of this line. In the comments for the disassembled code, the absolute address is given as "0x4005", which, as you can see, is the beginning address of this particular line of code. The most important part, though, is to understand the machine code: "20FE". "20" means a short branch always. "FE", though, is –2, and tells the microprocessor to subtract 2 from the current program counter. Since the program counter will have advanced by 2 while executing this command to address $4007, it will be moved back to $4005, where it will execute the command again ad infinitum.

Briefly look at the difference between the "bra *" line and the "**lbra ***" line. In this case, you'll notice that the relative offset is four nibbles: $FFFC, or -4, to cover the extra bytes required in this longer version of the command.

Look at the "**brset $4000,#1,forever**" line. In this command, if the LSB of the value in address location $4000 is set (the mask "#1" is the same as #%00000001), the program counter is supposed to go back to "forever", which is address $4007. (Incidentally, address $4000 contains $CF as seen in the program listing, so the branching condition will be TRUE.) Notice what the machine code says: "1E400001F9". "1E" is the op code for BRSET. "4000" is the address we want to check the contents of. "01" is the mask we're comparing against. And now for the offset: "F9" is only 8-bit, so we can only do short branches with this command. "F9" is –7, which takes the program counter back from $400E where it's sitting to $4007, the address of "forever".

Indexed – IDx, IDx1, IDx2, [IDx2], [D,IDx]

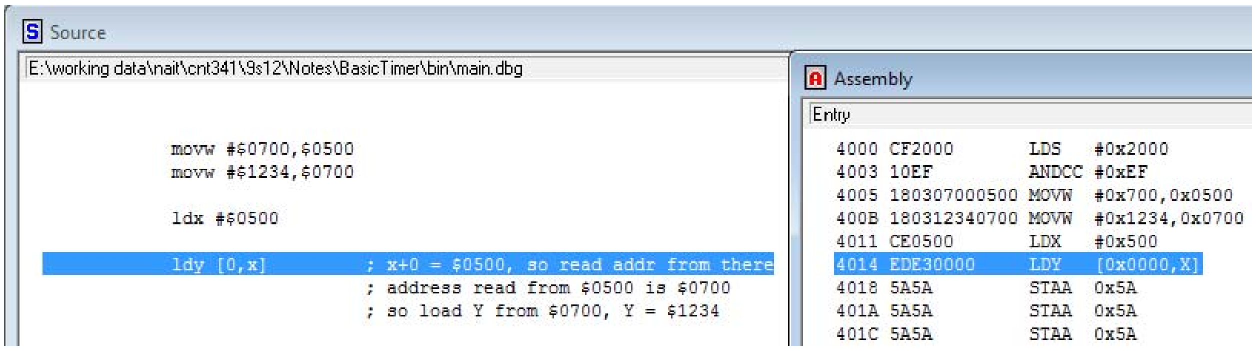
The S12XCPU chip has several forms of indexed addressing. The simplest form of indexed addressing uses X,Y,SP or PC ("x" in the form above) as a pointer. To this pointer (the value in the register) an offset is added. The offset is either a 5-bit signed offset (IDx), 9-bit signed offset (IDx1), 16-bit signed offset (IDx2), or the contents of accumulators A, B, or D. The assembler will automatically select the correct form of the instruction, as long as what you enter can be bashed into a valid instruction:



```
E:\working data\nait\cnt341\9s12\Notes\BasicTimer\bin\main.dbg

            ;bra next

            movw #$0102,$1000
            movw #$0304,$1002

            ldx #$1000

            ldaa 0,x       ; 5-bit
            ldaa $20,x     ; 9-bit
            ldaa $2000,x   ; 16-bit
            ldaa a,x       ; accumulator offset
```

```
4000 CF2000       LDS   #0x2000
4003 10EF         ANDCC #0xEF
4005 180301021000 MOVW  #0x102,0x1000
400B 180303041002 MOVW  #0x304,0x1002
4011 CE1000       LDX   #0x1000
4014 A600         LDAA  0x0,X
4016 A6E020       LDAA  0x20,X
4019 A6E22000     LDAA  0x2000,X
401D A6E4         LDAA  A,X
401F 5A5A         STAA  0x5A
```

The indexed-indirect addressing mode ([IDx2] and [D,IDx])allows either a 16-bit or D offset from X,Y,SP, or PC. In this mode the address formed from the offset is used as another address. The action of the instruction is on the target of this address:



```
E:\working data\nait\cnt341\9s12\Notes\BasicTimer\bin\main.dbg

            movw #$0700,$0500
            movw #$1234,$0700

            ldx #$0500

            ldy [0,x]          ; x+0 = $0500, so read addr from there
                               ; address read from $0500 is $0700
                               ; so load Y from $0700, Y = $1234
```

```
4000 CF2000       LDS   #0x2000
4003 10EF         ANDCC #0xEF
4005 180307000500 MOVW  #0x700,0x0500
400B 180312340700 MOVW  #0x1234,0x0700
4011 CE0500       LDX   #0x500
4014 EDE30000     LDY   [0x0000,X]
4018 5A5A         STAA  0x5A
401A 5A5A         STAA  0x5A
401C 5A5A         STAA  0x5A
```

NOTE: You can live a long, happy life not using most of the indexed addressing modes. However, they're there if you need them.

The indexing modes also offer pre/post increment/decrement options for the indexed addressing modes. These are typically leveraged by compilers, but you are free to look up their operation, should you feel ambitious.

You should get to know the direct indexing modes (no square brackets) very well. Here's a bit of practice.

```
            LDX       #Table
            LDAA      2,X
            BRA       *
            ORG       $C000
Table:      DC.B      $1E, $B6, $2F, $5A
```

After this code has run, *X* = $C000, and *A* = $2F. Make sure you can explain why before moving on.

Frequently-Used Instructions

In the Reference Guide, you will find a listing of all the possible instructions (the "Instruction Set") for the S12XCPU. You should look through this entire list to see what sorts of things you can do with this device.

Here are a few of the ones you will probably use extensively. As you go through this list, remember that references to "memory location" could refer to the byte or word directly following the Op Code (IMM mode) or a memory location elsewhere, accessed using any of the other addressing modes.

LDAA, ***LDAB***, and ***LDx*** – (where ***x*** could be ***D***, ***X***, ***Y***, or ***S***) puts the contents of a memory location into the selected accumulator or register. Remember that 8-bit registers will load a single byte and 16-bit registers will load two bytes – always the one you point to and the one immediately following it – in order to get the full 16 bits.

STAA, ***STAB***, and ***STx*** – (where ***x*** could be ***D***, ***X***, ***Y***, or ***S***) puts the contents of the selected accumulator or register into a memory location. Again, remember that 8-bit registers will store a single byte into the location you're pointing to, and 16-bit registers will store two bytes – one into the location you're pointing to and one into the one following it. If you forget this, you're in for a big surprise when you over-write a byte you didn't think you were going to affect.

CLR, ***CLRA***, and ***CLRB*** – all bits cleared in the selected accumulator or memory location.

DEC, ***DECA***, ***DECB***, ***DEx*** – (where ***x*** can be ***S***, ***X***, or ***Y***) subtracts one from a memory location or register.

INC, ***INCA***, ***INCB***, ***INx*** – (where ***x*** can be ***S***, ***X***, or ***Y***) adds one to a memory location or register.

BCC and ***BCS*** – branch to a specified location, based on the condition of the Carry flag

BEQ and ***BNE*** – branch based on the condition of the Zero flag

BGE, ***BGT***, ***BLE***, ***BPL***, ***BMI***, and ***BLT*** – branching decisions based on the comparison of signed numbers.

BHI, ***BLO***, ***BHS***, and ***BLS*** – branching decisions based on the comparison of unsigned numbers.

DBEQ and ***DBNE*** – compound instructions that decrement a register or memory location, then make a decision based on whether or not the result is zero.

ADDx and ***ADCx*** – (where ***x*** could be ***A***, ***B***, or ***D***) these add the contents of a memory location to the contents of the selected accumulator. If you use the "ADC" version, whatever is in the Carry flag of the CCR (0 or 1) will also be added in.

SUBx – (where ***x*** could be ***A***, ***B***, or ***D***) subtracts the contents of a memory location from the contents of the selected accumulator.

MUL – multiplies A by B and dumps the result in D.

There are five different division routines: ***FDIV***, ***EDIV***, ***EDIVS***, ***IDIV***, and ***IDIVS***. These are somewhat complicated to use, and will be explained when you need them.

ANDA and ***ANDB*** – these perform a bit-wise AND between the contents of the specified accumulator and the contents of a memory location (more later when we discuss masks).

ORAA and ***ORAB*** – these commands perform a bit-wise OR between the contents of the accumulator and the contents of a memory location.

EORA and ***EORB*** – performs a bit-wise Exclusive OR (XOR) between the accumulator and the contents of a memory location.

COM, ***COMA***, and ***COMB*** – performs a 1's complement inversion of each bit.

BITA and ***BITB*** – (bit test) performs an "AND" operation between the accumulator and the memory location, but doesn't affect the contents of either – only the Condition Code register is affected.

CBA – compares the A and B accumulators by subtracting B from A, and modifies the Condition Code Register accordingly – used to determine which is greater.

CMPA, ***CMPB***, ***CPx*** – (where ***x*** can be ***D***, ***S***, ***X***, or ***Y***) compares the selected register to memory by subtracting the contents of memory from the register, but doesn't change anything except the CCR.

LSL and ***LSLx*** – (where ***x*** could be ***A***, ***B***, or ***D***) performs a logical shift left, bringing in a "0" at the lowest bit and spitting the highest bit into the Carry flag of the CCR.

LSR and ***LSRx*** – (where ***x*** could be ***A***, ***B***, or ***D***) performs a logical shift right, bringing in a "0" at the highest bit and spitting the lowest bit into the Carry flag.

ROL, ***ROLA***, and ***ROLB*** – just like LSL, except that the contents of the Carry flag are brought in to the lowest bit instead of "0". Watch this command: it rolls 9 bits, not 8.

ROR, ***RORA***, and ***RORB*** – just like LSR, except that the contents of the Carry flag are brought in to the highest bit. Again, this rolls 9 bits, not 8.

BCLR – clears bits (ensures that bits are "0") according to which bits are set in a mask. Other bits remain unchanged. This involves ANDing the bitwise complement of the mask.

BSET – sets bits (ensures that bits are "1") according to which bits are set in a mask. Other bits remain unchanged. This involves ORing the mask.

CLC – clears the Carry flag in the CCR.

CLI – clears the Interrupt bit in the CCR, thereby enabling interrupts.

EXG, ***XGDX***, and ***XGDY*** – swaps the contents of two registers. Things get messy if you swap the contents of 8-bit and 16-bit registers!

TFR, ***TAB***, ***TBA***, ***TAP***, ***TPA***, ***TSX***, ***TXS***, ***TSY***, and ***TYS*** – moves the contents of one register into another without changing the first register's contents. Again, transferring the contents from an 8-bit register to a 16-bit or *vice versa* can produce unexpected results!

MOVB and ***MOVW*** – moves a byte (8-bit) or a word (16-bit) from one memory location to another. Frequently used in IMM/EXT mode, (e.g. MOVB #$3E,Table+1) this can also be used in EXT/EXT mode (e.g. MOVB Counter,Display) or various indexed (IDX) modes. Note: Don't confuse the "B" for "byte" in MOVB with Accumulator B!

PSHx and ***PULx*** – (where ***x*** could be ***A***, ***B***, ***C*** (for CCR), ***D***, ***X***, or ***Y***) places an item on the stack or takes it back off the stack, allowing you to use the stack as temporary storage.

JSR and ***RTS*** – jump to a subroutine and return from a subroutine. Be careful with these – they automatically involve placing the Program Counter (16-bit) on the stack and pulling it back off. If you follow a JSR with a branching statement instead of an RTS, you will quickly experience the pain of a stack overflow. Every JSR must have an RTS.

Masks and Bitwise Boolean Logic

Many of the instructions for the S12XCPU involve masks. A mask is an 8-bit or 16-bit binary pattern used to select individual bits in a register or memory location. In its original sense, the pattern "masked out" bits that weren't needed or wanted for a particular operation, leaving the significant ones "visible". We now use the term more generally for any pattern that allows us to operate on individual bits instead of the whole byte or word.

The various bitwise actions are as follows:

- SET the selected bits (i.e. make these bits HIGH, or logic 1).
- CLEAR the selected bits (i.e. make these bits LOW, or logic 0).
- TOGGLE the selected bits (i.e. LOW becomes HIGH, HIGH becomes LOW).
- BRANCH if the selected bit or bits is LOW.
- BRANCH if the selected bit or bits is HIGH.

It's important to distinguish between instructions or actions that affect an entire register or memory location and those that act on individual bits.

Commands affecting an entire register or memory location

LDAA/LDAB/LDD/LDX/LDY The register contents are replaced by the incoming data.

```
e.g.    LDAA #%10100011  ;A ends up containing 10100011
```

STAA/STAB/STD/STX/STY The memory contents are replaced by the outgoing data.

```
e.g.    STAA Counter     ;Counter ends up containing the contents of A
```

MOVB/MOVW The memory contents are replaced by the indicated data.

```
e.g.    MOVB #$F2,Counter ;Counter ends up containing F2.
```

CLRA/CLRB/CLR Each bit in the register or memory location is cleared to 0.

```
e.g.    CLR Counter      ;Counter ends up containing 00.
```

COMA/COMB/COM Each bit in the register or memory location is toggled (complemented).

```
e.g.    LDAA #%01011010
        COMA             ;A ends up containing 10100101
```

Commands affecting selected bits

ORAA/ORAB 1s in the mask SET the corresponding register bits; 0s have no effect.

```
e.g.    LDAA #%01011010
        ORAA #%11000000  ;A ends up containing 11011010
```

ANDA/ANDB 0s in the mask CLEAR the corresponding register bits; 1s have no effect.

```
e.g.    LDAA  #%01011010
        ANDA  #%11100111 ;A ends up containing 01000010
```

BSET 1s in the mask SET the corresponding bits in memory; 0s have no effect.

```
e.g.    MOVB  #%01100010,Counter
        BSET  Counter,%11000000 ;Counter ends up with 11100010
```

BCLR 1s in the mask CLEAR the corresponding bits in memory; 0s have no effect. This is essentially the same as ANDing with the complement of the mask.

```
e.g.    MOVB #%01100010,Counter
        BCLR  Counter,%11000000 ;Counter ends up with 00100010
```

Commands responding to selected bits

BRSET If, in a memory location, a bit or all of the bits selected by 1s in the mask are HIGH, program execution will branch to the indicated address. The most straightforward way to use BRSET is in response to a single bit.

e.g.

```
BRSET PT1AD1,%00010000,UpSw  ;Branches if the Up switch is pressed
```

BRCLR If, in a memory location, a bit or all of the bits selected by 1s in the mask are LOW, program execution will branch to the indicated address. Again, this is easiest to use in response to the condition of a single bit.

e.g.

```
BRCLR PT1AD1,%00010000,NotUp ;Branches if the Up switch is not pressed
```

Using Variables and Constants

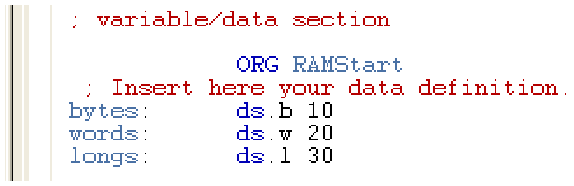
As you code more complicated tasks, you will find it increasingly difficult to juggle the few CPU registers you have to work with. The use of RAM-based variables is an easy sell, and will help you with code management when there are multiple states to manage.

Variables need to be stored in RAM, or they won't be variable. Variables are created within ORG sections that place the program counter in RAM. The skeleton file you're working with has a header to show you where you should place your variables, which will be after an "ORG" that places the variables in RAM.

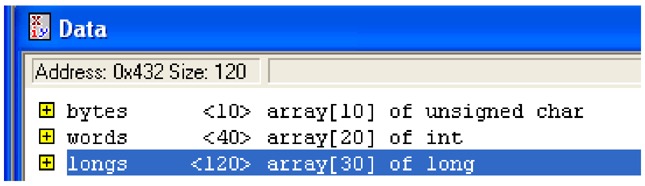
Variables are defined with a DS (Define Space) directive. There are three forms of the DS directive:

- DS.B reserve space for bytes (8-bits)
- DS.W reserve space for words (16-bits)
- DS.L reserve space for longs (32-bits)

The DS.*x* directive is followed by a count that indicates the number of elements to reserve. This number must range from 1 to 4096. The count is multiplied by the size of the type to determine the number of bytes that will be reserved for storage.



```
; variable/data section

            ORG RAMStart
 ; Insert here your data definition.
bytes:      ds.b 10
words:      ds.w 20
longs:      ds.l 30
```



```
Address: 0x432 Size: 120

bytes       <10>  array[10] of unsigned char
words       <40>  array[20] of int
longs      <120>  array[30] of long
```

Reserved space is not initialized, and will typically contain garbage. It is your responsibility to initialize all reserved space if your code requires it.

You will usually include a label for each DS directive, although it's not required. The label and reserved space together are loosely referred to as a 'variable'.

NOTE: You've been told this before, but it doesn't hurt to say this again: Your library subroutines may use variables internally, but because of the layout of the projects we are creating, the variables must be created in the main program file. Use of variables in a subroutine in a library will require that you clearly document the required variable names and initial values in the subroutine block.

Constants are not something intended to change as the program runs. Consequently, they should be in ROM. For ease of debugging, we'll use the address space starting at $C000.

Constants are defined with a DC (Define Constant) directive. There are three forms of the DC directive:

- DC.B byte-sized constants (8-bits)
- DC.W word-sized constants (16-bits)
- DC.L long constants (32-bits)

Since the data is constant, it must be defined at the time of assembly. Therefore, the DC command is followed by the data to be defined.

A typical application would be to store string data, as shown in the following example:

```
Name:       DC.B   "P. Ross Taylor",0
```

This will create a 15-byte field with an ASCII character in each byte (the last one is NULL), where the address indicated by "Name" will be the address of the character "P".

Here are some important things to keep in mind when using variables and constants in your code.

1. The Assembler interprets your assignment of a constant or variable label as a 16-bit address, and will replace subsequent references to that label with the 16-bit address.
2. Storing a single byte into a multi-byte variable changes only the first byte.
3. Storing a 16-bit value to a single-byte variable will change the contents of that address and the address following it. If that address is another variable, you will have over-written its contents (affectionately known as clobbering the next variable).
4. Storing to a constant in ROM changes nothing – why else would we call it a "constant"?
5. Calling something in RAM "constant" is a lie – don't do it! You could be fooled when debugging your program, as the Assembler will populate that location when you download your code. However, when you turn off the micro board, that information will be lost forever, and the code won't run properly the next time.
6. The Assembler lets you enter bytes into a constant or variable using a lot of different formats. For example:

   ```
   Str: dc.b   $48,'i',$64,"den",$20,"Me",%01110011,115,$60+1,'g',$67-2
   ```

   …will contain the ASCII characters "*Hidden Message*". See if you can determine how each of the characters is interpreted from what is given by the Assembler. You may find some of these techniques useful. For instance:

   - Putting an ASCII character in single quotes (*'i'*) tells the Assembler to treat this as an ASCII character.
   - Putting multiple ASCII characters in double quotes (*"den"*) tells the Assembler to put each of the ASCII characters into sequential address locations.
   - Characters can be entered in binary, decimal, hexadecimal, or even octal form.
   - The Assembler will even do calculations "on the fly" to arrive at a value to store in an address location.
7. The X and Y registers are called "index registers" because you can use them as pointers to the beginning of a memory space (like a multi-byte constant or variable).
8. Use a pound sign (#) to load the address of a variable or constant into a 16-bit register, usually (but not exclusively) an index register (*X* or *Y*).
9. You can find a byte at a particular offset from the address contained in an index register. This is done using Indexed Addressing Mode, for example

   ```
   LDX   #Str              ;X now points to the first address of Str
   LDAA  2,X               ;A now holds the third char in Str (zero based)
   ```

10. The index registers can point to any memory location (not just the start of a variable), so you can crawl through a multi-byte variable or constant using INX or INY.

    ```
    INX                     ;following the above code, X points to Str+1
    ```

11. You can't load an address into an 8-bit register. You will only get half of the address, and certainly not the contents of that address, so trying to work with that will produce really bizarre results. So don't try `LDAA #Str` – it won't work.
12. If you don't use a pound sign, you will load the contents of the memory location represented by the variable. If you load an 8-bit register, you will get the single byte from that memory location (`LDAA Str` puts 'H' into *A* in our previous example). If you load a 16-bit register, you will get two bytes: the one you pointed to and the one following it. (`LDD Str` puts 'H' into *A* and 'i' into *B* in our previous example).

Programming in C

Every time you've started a new project, you've unchecked "C" and checked "Absolute Assembly" instead. You probably don't even think about it anymore; but what if ….

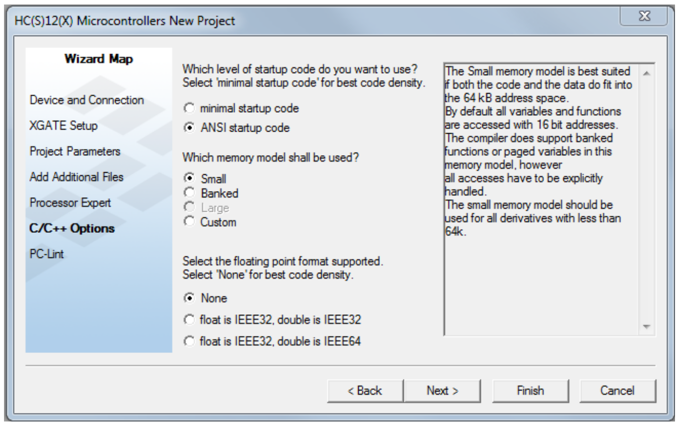
C is a much older language than the C# you've been working with. However, although Microsoft has tried to cloak the basics in Orwellian doublethink, you will probably find that there are some transferable concepts, fundamental operations, and syntax similarities that will make learning C for the 9S12X fairly easy. Unless your instructor has chosen to approach the order of this course differently, you should, at this point, have learned the fundamentals of programming in S12XCPU Assembly Language, which is tightly linked to an understanding of the operation of the microprocessor and its peripherals. Programming in C also requires a clear understanding of the operation of the microcontroller, particularly its registers; however, the C cross-compiler used by Code Warrior is capable of handling many things itself so you don't have to worry about all the details.

The following pages show how to start a C project, how to write and run simple code, how to write and use "Functions" (these are what C# architects decided to refer to as "Methods"), and how to write, include, and use uncompiled code libraries.

Setting Up an ANSI C Project

Start a new project, following the steps below:

1. Follow the usual steps to start a new project, including selecting the appropriate derivative of the 9S12X, the BDM interface you're using, and "Single Core".
2. Leave the "C" box checked, set the appropriate "Location" (your Projects folder), and give your project a name. Don't hit "Finish", as there are more screens coming.
3. When you get to the "C/C++ Options" page in the Wizard Map, select "ANSI startup code" and the "Small" memory model. If your program doesn't have to do anything mathematical, leave floating point format as "None". If, however, you want to do floating point math (i.e. fractional values) instead of just working with integers (not to be confused with *int* declarations), you should probably check "float is IEEE32, double is IEEE32". (This option will consume a lot more memory when generating code and will run more slowly, but you probably won't run out of room. Timing might be more of an issue.) Now you can click "Finish".



Which level of startup code do you want to use?
Select 'minimal startup code' for best code density.

minimal startup code [✓ ANSI startup code]

Which memory model shall be used?

[✓ Small] Banked Large Custom

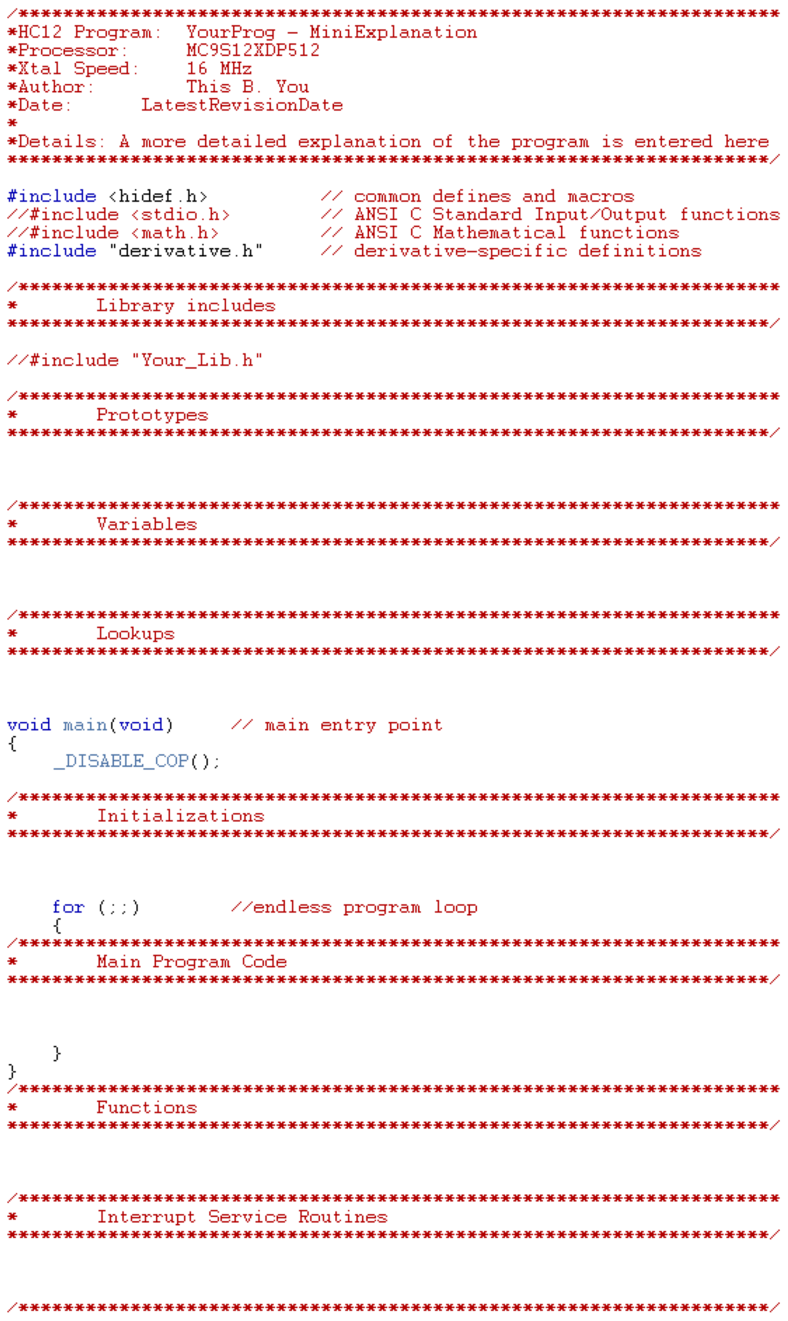
Select the floating point format supported.
Select 'None' for best code density.

[✓ None] float is IEEE32, double is IEEE32 float is IEEE32, double is IEEE64

The Small memory model is best suited if both the code and the data do fit into the 64 kB address space.
By default all variables and functions are accessed with 16 bit addresses.
The compiler does support banked functions or paged variables in this memory model, however all accesses have to be explicitly handled.
The small memory model should be used for all derivatives with less than 64k.

4. Open the "main.c" program. It will contain some basic code (a bit less annoying than the code they inserted in "main.asm", but still not entirely useful. Instead, create a "skel_C.txt" file like the one on the following page, replace the text in "main.c" with it, and tidy up the header information.

ANSI C Skeleton File



```
/********************************************************************
*HC12 Program:  YourProg - MiniExplanation
*Processor:     MC9S12XDP512
*Xtal Speed:    16 MHz
*Author:        This B. You
*Date:      LatestRevisionDate
*
*Details: A more detailed explanation of the program is entered here
********************************************************************/

#include <hidef.h>          // common defines and macros
//#include <stdio.h>        // ANSI C Standard Input/Output functions
//#include <math.h>         // ANSI C Mathematical functions
#include "derivative.h"     // derivative-specific definitions

/********************************************************************
*       Library includes
********************************************************************/

//#include "Your_Lib.h"

/********************************************************************
*       Prototypes
********************************************************************/


/********************************************************************
*       Variables
********************************************************************/


/********************************************************************
*       Lookups
********************************************************************/


void main(void)     // main entry point
{
    _DISABLE_COP();

/********************************************************************
*       Initializations
********************************************************************/


    for (;;)        //endless program loop
    {
/********************************************************************
*       Main Program Code
********************************************************************/


    }
}
/********************************************************************
*       Functions
********************************************************************/


/********************************************************************
*       Interrupt Service Routines
********************************************************************/


/********************************************************************/
```

There are a number of ways to create an endless loop in ANSI C, including *while(1){code}*. However, we've chosen to use the endless *for (;;){code}* loop as shown in the skeleton file above because it doesn't generate any compiler warnings.

Notice that *Variables* and *Lookups* appear to be in the same memory space, which would have to be in RAM. However, in ANSI C, both of these can be initialized or pre-loaded, which means their values would somehow have to be in ROM. In reality, the C compiler will store the initialization values in ROM, and will create a working copy in RAM on start-up so that the values can subsequently be changed under programmatic control. (The same can be done when programming in S12XCPU Assembly Language, but the process has to be written into the program, and is therefore much more involved.)

Switches and LEDs with ANSI C

Since you've already gained some experience working with the push-button switches and the LEDs connected to PT1AD1 of your microcontroller, this makes a good jumping point into writing programs in ANSI C.

In order to initialize PT1AD1, you need to set the LED-connected pins to OUTPUTS, set the Switch-connected pins to INPUT, and digitally-enable the switch-connected pins. You probably also want to initialize the conditions of the LEDs to ALL OFF. Here's a bit of ANSI C code that performs these operations:

```
DDR1AD1=0b11100000;      //LEDs as outputs, Switches as inputs
ATD1DIEN1=0b00011111;    //Digitally-enable the Switch inputs
PT1AD1&=0b00011111;      //Turn off all LEDs as initial condition
```

Notice that we can overwrite all eight bits in a register using the "=" operator. In the example above, the values have been entered as binary values, because that makes the best sense for bitwise operations. However, the first command could have been "DDR1AD1=0xE0" for hexadecimal or "DDR1AD1=0340" for octal or "DDR1AD1=224" for decimal – all of these options would have made the LED pins outputs and the Switch pins inputs. The cross-compiler is smart enough to convert any numeric representation into binary for the microcontroller, so pick the format that makes the most sense to you as a human-programmer. There are times when the decimal representation of a number makes most sense. For example, "NewDozen +=12" probably makes more sense to you than "NewDozen+=0b00001100", "NewDozen+=0x0C" or "NewDozen+=014".

If we only want to change some of the bits, we use bitwise operations like "&", "|", or"^" (AND, OR, or EOR). In the case above, we wanted to turn off the LEDs, so ANDing with 0 performs that function, whereas ANDing with 1 has no effect. If we wanted to turn on or set particular bits, we would OR the desired bits in the register with 1s, whereas ORing the other bits with 0 has no effect.

Functions

In ANSI C, a Function (a.k.a. Subroutine or Method in other languages) requires a "Prototype" or "Declaration", defined prior to the execution of any code. The prototype declares the *type* of what is returned from the function and the *types* of any parameters passed to the function. Although you can choose the variable names for parameters passed to the function in the prototype, this is not necessary. The prototype often looks like the first line in the actual function, just terminated with a semicolon. The following are some examples:

void SwLED_Init(void);

char SwCk(void);

void LEDOut(char LEDs);

void LEDOut2(char);

unsigned int TwoNumSum(unsigned char X, unsigned char Y);

The skeleton file provided has a section at the top for you to put your prototypes.

The function itself starts with a header much like the prototype, but with the variable names that will actually be used in the function, if these were not specified in the prototype. The "definition" of the function is contained between curly brackets {} (affectionately referred to as "chicken lips" in NAIT's CNT department).

If a value is to be returned from the function, a "return <value>" statement is required.

Libraries of Functions

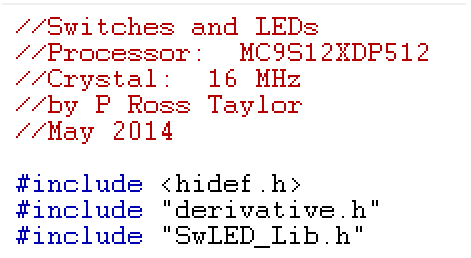
As with your work with S12XCPU Assembly Language, you can create libraries of commonly-used functions. However, the process is quite different.

To begin with, you will need two files for each library: a header file (.h) and an uncompiled code library (.c). The header file contains all of the prototypes for the functions, with the definitions (actual code) appearing in the uncompiled code library.

When you create a new project, you will need to do three things:

- Include the ".c" file in the "Sources" section of the project browser window.
- Include the ".h" file in the "Includes" section.
- Add an #include "libname.h" line to the "Library includes" section of the skeleton.

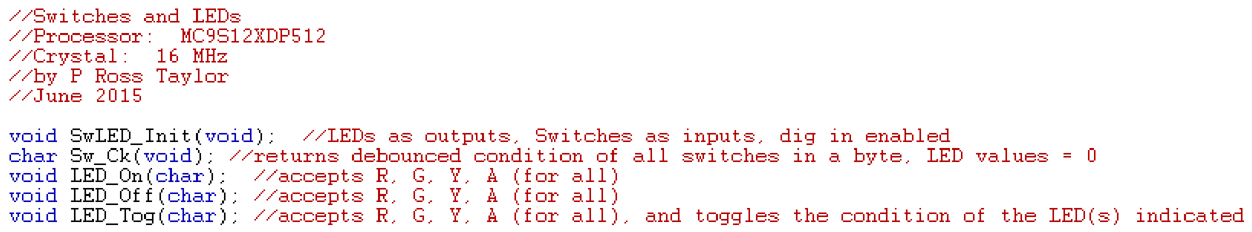
The ".c" file itself needs to start with "include" statements. Here's a screen-shot of the beginning of this author's "SwLED_Lib.c" library:



```
//Switches and LEDs
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//May 2014

#include <hidef.h>
#include "derivative.h"
#include "SwLED_Lib.h"
```

Notice the different punctuation: the <hidef.h> reference is to one of the standard ANSI C compiled libraries, whereas the libraries in quotes are uncompiled libraries. The "derivative.h" file points to the "mc9s12xdp512.h" file that contains all the definitions of the labels for the registers in the version of the 9S12 we're using.

You will be required to write a library to match the contents of the following "SwLED_Lib.h" header file:



```
//Switches and LEDs
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//June 2015

void SwLED_Init(void);  //LEDs as outputs, Switches as inputs, dig in enabled
char Sw_Ck(void); //returns debounced condition of all switches in a byte, LED values = 0
void LED_On(char);  //accepts R, G, Y, A (for all)
void LED_Off(char); //accepts R, G, Y, A (for all)
void LED_Tog(char); //accepts R, G, Y, A (for all), and toggles the condition of the LED(s) indicated
```

In the functions that accept a colour parameter, you will need to pass the value as an ASCII character, which requires the use of single quotes: e.g. 'R'.

To begin with, you won't need to build the "Sw_Ck()" function, as you need to learn a bit more about switch management before you can deal with that one.

Summary

You have now been provided with a bare minimum of what it takes to program in ANSI C. With your experience with other programming languages, particularly C#, you should be able to play around with ANSI C quite productively, and you will learn more of what the language looks like and what it can do as this course progresses.

Numeric Manipulation

A review of bit basics is prudent at this point, as an understanding of binary and hexadecimal will be assumed throughout the rest of this course.

Understanding Base 10

Base 10, or decimal, is a good radix to begin with, as you are familiar with it. We know that each digit contributes the digit value * 10^n, where n is the zero-based index of the digit, working right to left. Consider the number 343895_{10}:

Digit	3_{10}	4_{10}	3_{10}	8_{10}	9_{10}	5_{10}
Position Value	10^5	10^4	10^3	10^2	10^1	10^0
Digit Value	300000_{10}	40000_{10}	3000_{10}	800_{10}	90_{10}	5_{10}

$343895_{10} = 300000_{10} + 40000_{10} + 3000_{10} + 800_{10} + 90_{10} + 5_{10}$
$343895_{10} = 343895_{10}$

This pattern seems obvious for base 10, but works for base 2 (binary) and base 16 (hexadecimal) as well.

Converting Binary to Decimal

In binary the number is valued as the sum of each digit * 2^n, where n is the zero-based index of the digit, working right to left. Consider the number 100101_2:

Digit	1_2	0_2	0_2	1_2	0_2	1_2
Position Value	2^5	2^4	2^3	2^2	2^1	2^0
Digit Value	32_{10}	0_{10}	0_{10}	4_{10}	0_{10}	1_{10}

$100101_2 = 32_{10} + 4_{10} + 1_{10}$
$100101_2 = 37_{10}$

The system shown above is called "Weighted Sum of Powers".

Converting Hexadecimal to Decimal

Hexadecimal is no different, other than including A-F as digits to allow each hex digit to represent one of 16 different values.

Decimal	**Binary**	**Hexadecimal**
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Consider the number $3D5F2A_{16}$, converted as shown below using weighted sum of powers:

Digit	3_{16}	D_{16}	5_{16}	F_{16}	2_{16}	A_{16}
Position Value	16^5	16^4	16^3	16^2	16^1	16^0
Digit Value	3145728_{10}	851968_{10}	20480_{10}	3840_{10}	32_{10}	10_{10}

$3D5F2A_{16} = 3145728_{10} + 851968_{10} + 20480_{10} + 3840_{10} + 32_{10} + 10_{10}$
$3D5F2A_{16} = 4022058_{10}$

Converting Hexadecimal to Binary

Converting hex numbers to binary and vice versa is nice and easy, as each hex digit can be converted to a nibble. You may use the lookup table above, or your brain, to do the conversion. Using this technology, the hex number above ($3D5F2A_{16}$) could easily be converted to binary. Always remember to work right to left, and strip any leading zeros on the result (unless you want to show a specified number of bits in your result, regardless of what they are):

Hex	3_{16}	D_{16}	5_{16}	F_{16}	2_{16}	A_{16}
Binary	0011	1101	0101	1111	0010	1010

$3D5F2A_{16} = 001111010101111100101010_2$

or

$3D5F2A_{16} = 1111010101111100101010_2$

Converting Binary to Hexadecimal

Converting binary to hex requires that you work right to left 'snapping' the binary digits into nibbles, padding the left-most digits with zeros to fill the final nibble, if necessary.

For example, convert 1101010010101011010111_2 to hexadecimal:

Binary	1101010010101011010111_2					
Nibbler	0011	0101	0010	1010	1101	0111
Hexadecimal	3_{16}	5_{16}	2_{16}	A_{16}	D_{16}	7_{16}

$1101010010101011010111_2 = 352AD7_{16}$

8 Bit Arithmetic

You will principally be concerned with 8 bit numbers while coding. Mastery of all that is 8 bit needs to become part of your mental fabric – stat. Typically when you look at a binary or hexadecimal number, you assume it is unsigned. The fact that binary and hexadecimal numbers have no sign notation for 'negative' contributes to this. There are times when numbers need to be interpreted as signed, and binary numbers may be 2's complement coded to manage a signed value. Please note, before we get too far here, that there is no way to determine if a binary number is intended to be signed or unsigned – it is entirely up to context and interpretation. The S12XCPU has instructions that will assume that an operand is signed, and will interpret the binary number that way. These instructions are *fairly* clearly marked.

Binary numbers that are interpreted as being signed consider the most significant bit as contributing a negative value. This means that for an 8 bit number, the most significant bit will contribute -128 (-2^7), if it is set:

Bit Pattern	Hex Value	Unsigned Decimal Value	Signed Decimal Value
%00000000	$00	0	0
%10000000	$80	128	-128
%11111111	$FF	255	-1
%01111111	$7F	127	127

From this, we can glean a couple of important points:

- Representation of –0 is not possible
- The MSB directly represents the sign of the number (but not as a fundamental flag)

Working with 2's Complement

To code a number as 2's complement, you take the 1's complement and add 1.

NOTE: You only code negative numbers in 2's complement – if a number is positive and fits the signed range, then you need do nothing.

Consider the number -94_{10}. First, determine the binary representation of the number:

94	%01011110
Next, take the 1's complement:	%10100001
Add 1	%10100001
	+% 1

	%10100010

- OR -

94	%01011110
Start at the right and copy bits until you encounter a 1, then invert the rest:	%101000$\underline{10}$

2's complement is used to resolve subtraction with addition. You use 2's complement form to effectively flip a negative sign to a positive sign. This only has an effect on negative numbers. For example, consider the problem of 15 – 6:

```
15 = $F = %00001111
 6 = $6 = %00000110
```

Because it's negative, convert the 6 to 2's complement, then add to the 15:

```
 %00000110 ($06)
 %11111010 ($FA)

 %00001111 +
 %11111010
----------
%100001001 (discard overflow = $9)
```

So… 15 – 6 = 9 apparently…

If you were to try a problem like 6 – 15, you would find that the 15 needs to be converted to 2's complement, followed by addition:

```
 %00001111 ($0F)
 %11110001 ($F1)

 %00000110 +
 %11110001
 ---------
 %11110111 (bit 7 set, so negative; take 2's complement to get the value)
 %00001001 (answer is –9)
```

So… 6 – 15 = –9…

This, of course, would work exactly the same way if the problem was –15 + 6.