

Topic 2 – Assembly Language 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
- 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 outcomes will be partially 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 and assembly language programming you learn as you complete this topic.

Connection Activity

You have now learned enough about the 9S12 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.

Documentation and Comments

Take a look at the “Main.asm” file you’ve been working with. At the top, there’s a box of comments that starts with “This stationery serves ...”. Not terribly meaningful in terms of the programs you’ve been writing, is it? You should replace this box with a description of your program, and you should be writing comments that reflect the operation of your code.

One useful idea is to have a “skeleton” file that you use each time you start a new project. The easiest way to use a skeleton file is to copy and paste the text from it into “Main.asm” before you start writing code. 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.

```

;*****
;* HC12 Program:   YourProg - MiniExplanation                      *
;* Processor:      MC9S12XDP512                                    *
;* Xtal Speed:     16 MHz                                          *
;* Author:         YourNameHere                                    *
;* 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
```

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.

```
;*****
;*                               *
;*           Hex2BCD             *
;*                               *
;*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 Hex2BCD” 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  #$E0           ;load accumulator A with hex E0
Good: LDAA  #$E0           ;ready to initialize Port AD Data Direction Register
```

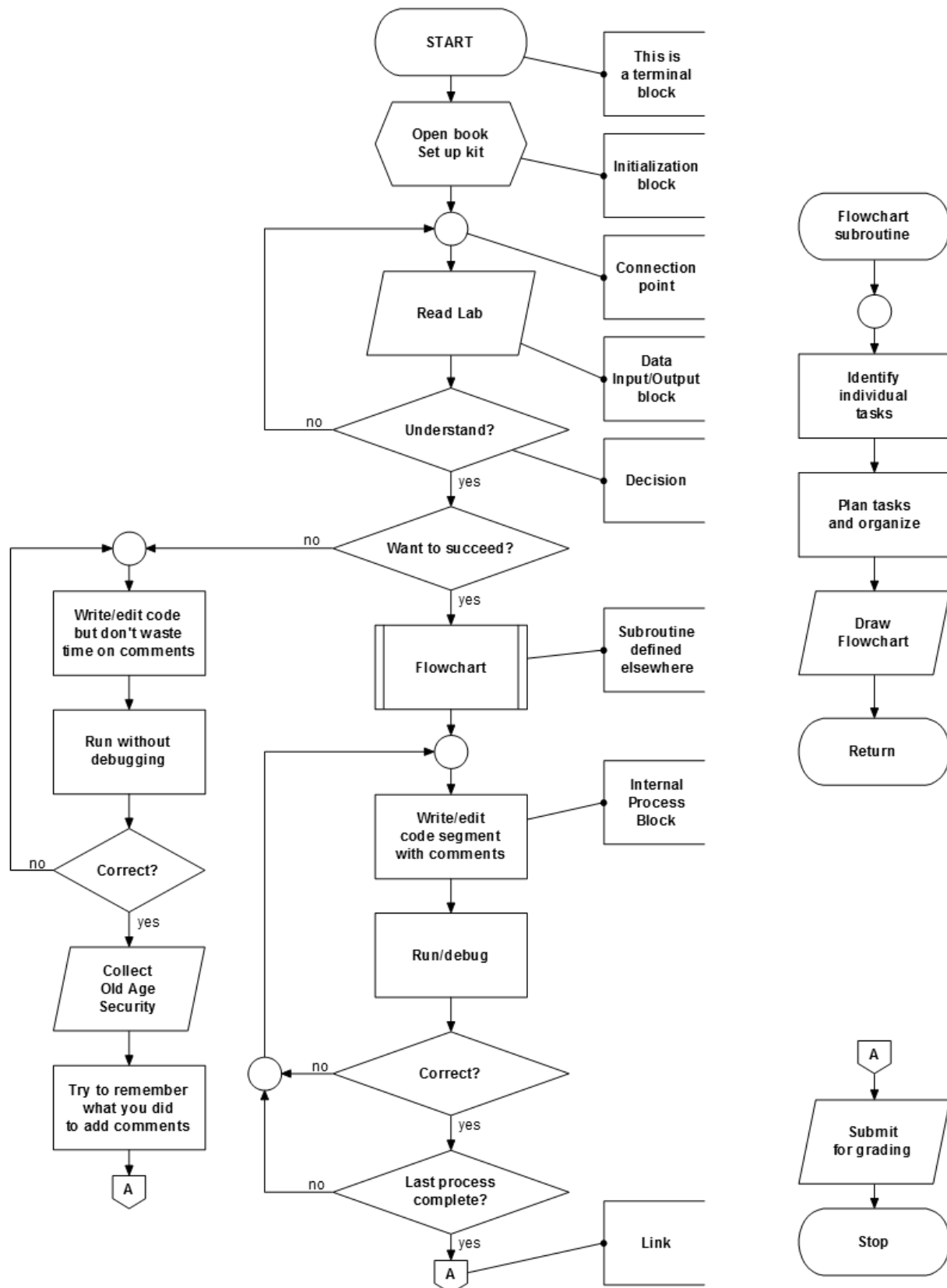
Don’t type pages of code and then try to go back and insert comments. The comments are there to help you keep track of what you’re doing as you are coding just as much as to help win your instructor’s favour!

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    SW_LED_Init
Flash:      JSR    RedLEDOn
            JSR    lmsDelay
            JSR    GrnLEDOn
            JSR    lmsDelay
            JSR    AllLEDOff
            JSR    lmsDelay
            BRA    Flash
```

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

Flowcharting



Your subsequent work in this course will be significantly more complex than what you've been doing up to this point. In order to succeed, you need to pre-plan your work. The flowcharting sequence and recognizable blocks, like those 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 we learn how to write 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 cannot 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 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: that kind of stuff should be handled by your main program.
- Don't change library subroutines after you've used them – previous programs won't work anymore!
- 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, make sure you put a note in the header reminding yourself or someone else using your subroutine that the variable needs to be declared in the Main program. (We'll talk more about variables later.)

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.

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" statement **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, delete everything except the subroutines, 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.

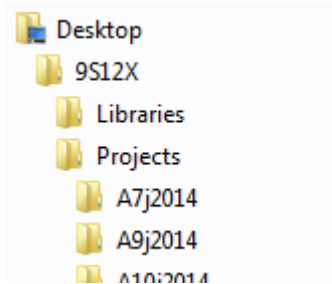
The more difficult part is deciding what you want to have in a library. You should collect together subroutines that are related, and are therefore likely to be used for a particular type of program. For example, you will be doing a lot of work with the Serial Communication Interface (SCI). It makes 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.

The libraries you'll end up with by the end of this course will include the following:

```
SW_LED_Lib.inc  
Misc_Lib.inc  
SevSeg_Lib.inc  
LCD_Lib.inc  
SCI0_Lib.inc  
PWM_Lib.inc
```

Misc_Lib.inc will contain routines that could be used in a number of types of programs – things like Hex2Asc, Hex2BCD, etc.

You will need to 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.

I would recommend storing your skeleton file in the "9S12X" directory so you can access it every time you start a project.

Of course, each time you start a project, you should give it an informative name (i.e. *not* Project1), and store it in the "Projects" folder. The CodeWarrior IDE will create the folder for that project and will build all the associated subfolders and files inside the project folder.

9S12 Addressing Modes

By now you have had some exposure to assembly language and basic program creation. It is time to look at addressing modes so that the full benefit of instruction flexibility can be realized.

Addressing modes define what memory the instruction will operate on. Each instruction offers one or more addressing modes. The 9S12 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. Some examples:

Learning Exercise 1. In the Reference Guide, locate the following Inherent Commands: ABA, INX, INY. What do these commands "work on"? Do they require any kind of address to work?

Learning Exercise 2. Locate the TFR instruction. What does this command "work on"? Why is it an Inherent command?

Learning Exercise 3. The following code snippet contains a number of Inherent commands. Notice how the Assembler interprets each – no reference to memory addresses.

The screenshot shows an assembly editor with two panes. The 'Source' pane on the left contains the following code:

```

nop
aba
inx
mul
tfr x,y

```

The 'Assembly' pane on the right shows the corresponding machine code instructions:

```

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
400C E3E3      STAA  0x53

```

The instruction at address 4005, 'NOP', is highlighted in blue in both panes.

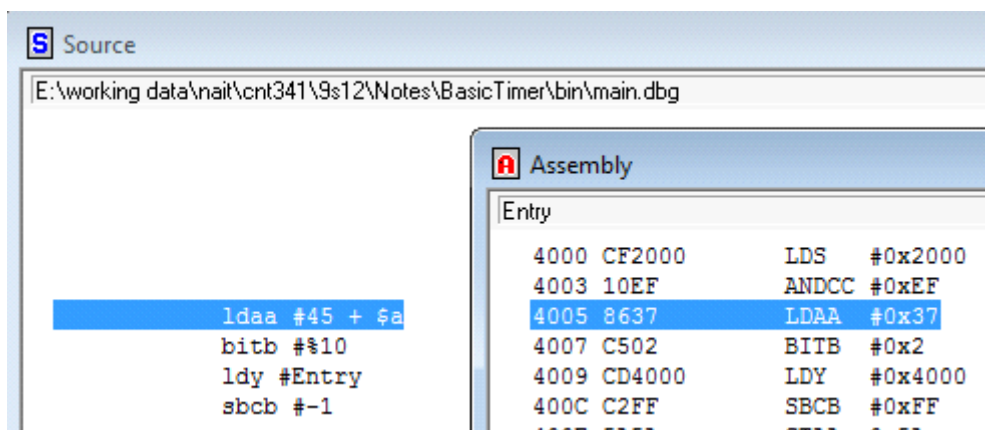
Immediate - IMM

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.

Learning Exercise 4. In the Reference Guide, locate the LDAA command. Notice that the first of eight forms of this command has, as its addressing mode, IMM. Also notice that it says the Source Form is "LDAA #opr8i", and the Machine Coding says "86 ii". All of the little "i" references mean this is all about immediate addressing, which means the assembler will grab the byte (or, for a 16-bit register, the word) immediately following the command as the data it will act upon. What do you think "opr8" means?

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 soon.

Learning Exercise 5. Examine the following code snippet, which contains a number of commands in Immediate Addressing mode. Notice how the Assembler interprets each command, and what it will work on.



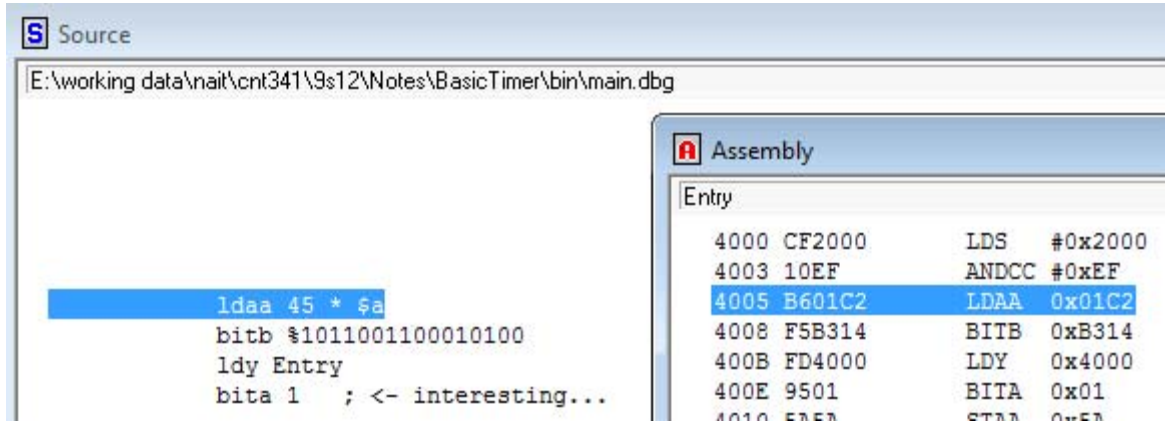
Source	
E:\working data\nait\cnt341\9s12\Notes\BasicTimer\bin\main.dbg	
ldaa #45 + \$a	
bitb #\$10	
ldy #Entry	
sbc b #-1	

Assembly	
Entry	
4000 CF2000	LDS #0x2000
4003 10EF	ANDCC #0xEF
4005 8637	LDAA #0x37
4007 C502	BITB #0x2
4009 CD4000	LDY #0x4000
400C C2FF	SBCB #0xFF
400F E3E3	STAB 0xE3

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:



Learning Exercise 6. Use the Op Codes shown in the “Assembly” window for the code snippet from \$4005 to \$400F, along with the Reference Guide, to determine the addressing modes implemented for each instruction.

We can use labels to define constants (typically in ROM) and variables (typically 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

```

Learning Exercise 7. Indicate *what* will be *where* after executing this code.

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 to 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, on the 9S12 in its default configuration, there is not a lot of call for direct addressing, since the “first page” of memory references internal microcontroller 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 9S12 supports long and short branching. The operand(s) in a branch instruction form a signed offset that participate(s) 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.

The screenshot shows a debugger interface with two windows: 'Source' and 'Assembly'.

Source Window: Displays assembly code with labels. The code is as follows:

```

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

Assembly Window: Displays the disassembled code with comments indicating absolute addresses. The code is as follows:

```

Entry
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, *0xFFFFFFFF ;abs = 0x4007
400E 2100    BRN  *+0x2      ;abs = 0x4010
4010 20FC    BRA  *0xFFFFFFFF ;abs = 0x400E
4012 1820FFFC LBRA  *+0x0      ;abs = 0x4012
4016 5A5A    STAA 0x5A
  
```

Learning Exercise 8. By comparing the “Source” window to both the machine language and disassembled code in the “Assembly” window, explain why the branching instructions are called “Relative”.

Learning Exercise 9. Notice the comments created by the disassembler. What do you learn about the use of labels (e.g. “forever”, “backward”, “forward”)?

Learning Exercise 10. What is the difference between “bra” and “lbra”?

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

The 9S12 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:

The screenshot shows an IDE with two panes. The 'Source' pane on the left contains assembly code. The 'Assembly' pane on the right shows the corresponding machine code instructions.

```

Source
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

Assembly
Entry
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:

The screenshot shows an IDE with two panes. The 'Source' pane on the left contains assembly code. The 'Assembly' pane on the right shows the corresponding machine code instructions.

```

Source
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

Assembly
Entry
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.

LDX	#Table
LDAA	2,X
BRA	*
ORG	\$C000

Table:	DC.B	\$1E, \$B6, \$2F, \$5A
--------	------	------------------------

Learning Exercise 11. Indicate *what* will be *where* after executing this code.

Frequently-Used Instructions

In the Reference Guide, you will find a listing of all the possible instructions (the “Instruction Set”) for the 9S12. You should look through this entire list to see what sorts of things you can do with this device. In fact ...

Learning Exercise 12. Mark whatever you use as a day-timer to remind you to go through the Instruction Set for the 9S12, as found in the Reference Guide. Oh, and you can't use the excuse that you left it in your locker, because it's available online – either from the Moodle site for this course or directly from Freescale.

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, this can also be used in EXT/EXT mode or various indexed (IDX) modes.

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.

Learning Exercise 13. Create a program from scratch (use the skeleton file described earlier) that will look for switch presses on the left, centre, and right switches. When one of these switches is pressed, you will turn on the corresponding LED (by physical position). The LED will remain on, regardless of what happens to the switch after it is pressed. Once all three switches have been pressed, all three LEDs will remain on for about 3 seconds, then the program will reset to the startup state. Use comments, and, for repetitive tasks, write subroutines.

Masks and Bitwise Boolean Logic

Many of the instructions for the 9S12 involve masks. A mask is an 8-bit or 16-bit binary pattern used to select bits in a register or memory location.

In some cases, we want to turn specific bits off while leaving others unaffected; in other cases, we want to turn specific bits on while leaving others unaffected.

Some instructions require a mask as part of the operand. In these cases, a "1" means you want to affect that particular bit, whereas a "0" means don't make any change to the specified bit.

In other cases, we need to manipulate bits in code, using our own masks. In these cases, the result is determined by bitwise Boolean logic.

Learning Exercise 14. Answer the following questions about masks. In each case, assume that DDR1AD1 initially contains the byte \$53. What would the contents of DDR1AD1 be after each of the following?

1. `MOVB #%11100000, DDR1AD1`
2. `BSET DDR1AD1,%11100000`
3. `BCLR DDR1AD1,%00011111`
4. `LDAA #%11100000`
`STAA DDR1AD1`
5. `LDAA DDR1AD1`
`AND #11100000`
`STAA DDR1AD1`
6. `LDAA DDR1AD1`
`ORA #11100000`
`STAA DDR1AD1`

There are times when we want direct control over all the bits in a register, and times when we must not change certain bits. This determines whether we choose to use "MOVB" or whether we will use "BSET" or "BCLR". (Typically, with the 9S12, there's limited use for manipulating registers using "AND" or "OR". That's something you need to know about in case you run into a different microcontroller that doesn't have the extensive set of commands available for the 9S12.)

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 $\times 10^n$, 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 $\times 2^n$, 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}

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

$$100101_2 = 37_{10}$$

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 $3D5F2A_{16}$:

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 9S12 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 <u>1</u> 0

2's complement is used to resolve subtraction, oddly, 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 this is a negative result; take 2's complement to produce sign)

00001001 (answer is -9)

So... $6 - 15 = -9$...

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

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.

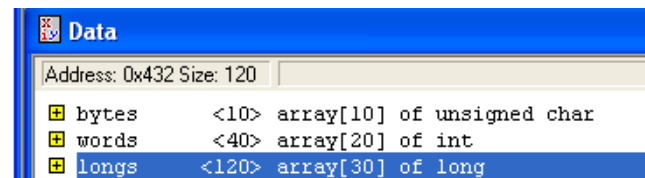
Variables need to be stored in RAM, or they won't be variable. Variables are created within ORG sections that place the program counter in RAM. The skeleton file you've hopefully created by now has a header to show you where you should place your variables, which will be after an "ORG" that places the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



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"

This will create a 14-byte field with an ASCII character in each byte, where the address indicated by "Name" will be the address of the character "P".

Learning Exercise 15. Do the following to help you grasp how variables and constants work.

1. Create a new project using the Skeleton file.
2. In the "Variables" space (i.e. after ORG RAMStart), enter the following:

```
Variable1:  ds.b  2
Variable2:  ds.b  1
FailConst:  dc.b  "This is going to disappear"
```

3. In the "Constants" space (i.e. at \$C000 and following), enter the following:

```
GoodStr:    dc.b  "This will not disappear"
AltStr:     dc.b  $48,'i',$64,"den",$20,"Me",$73,115,$60+1,'g',$67-2
FailVar:    ds.b  1
```

4. In the "Main" space enter the following, but don't try to run it as proper code (we'll step through it to see what happens):

```
CLR        Variable1
CLR        Variable2
LDX        #GoodStr
LDAA       #GoodStr
LDAA       GoodStr
LDY        GoodStr
LDAA       0,X
LDAB       5,X
LDD        5,X
STD        Variable1
STAB       Variable1
STAA       Variable1+1
STAB       Variable2
STD        Variable2
STAA       FailVar
INX
LDAA       0,X
BRA        *
```

5. Run the Assemble/Debug routine to burn the above into your microcontroller and to open the Debugger. You will get one warning, but ignore it for now. (Don't ignore it if you see it in your own code! It really is an error, and I want you to see why.)
6. In the "Data" window, you will see each of the constants and variables you defined above. Answer the following:
 - a. Why does Variable1 have an array of 2, whereas Variable2 doesn't?
 - b. In the "Data" window, expand "FailConst" and "GoodStr". What do you see? What do you learn about dc.b?
 - c. In the "Memory" window, locate \$C000. Compare what appears in the actual bytes (middle section) to what you see in the decoded section (right side). Explain these in terms of what you know about ASCII coding.

- d. Explain the contents of AltStr, byte by byte, by comparing to the code you entered. You may want to switch the format back and forth between "Symbolic" and "Hex".
 - e. Turn off your board, wait a while, then turn it back on. Now, refresh the Data window. What happened to "FailConst" and "GoodStr"? Why?
7. Step through the first three lines of code. You'll probably want to observe what happens to Variable1 and Variable2, although the focus of this question is on X.

```
CLR      Variable1
CLR      Variable2
LDX      #GoodStr
```

What appears in X, and why?

8. After LDAA #GoodStr, what appears in A, and why? Go back to the warning the Assembler generated: What does it mean?
9. Explain the contents of A after LDAA GoodStr.
10. After LDY GoodStr, what appears in Y, and why? Is this likely to be what you would want to have in Y?
11. After LDAA 0,X, what appears in A, and why?
12. After LDAB 5,X, what appears in B, and why?
13. After LDD 5,X what appears in A and B, and why?
14. After STD Variable1, what appears in Variable1?
15. After STAB Variable1 and STAA Variable1+1, what appears in Variable1? Why?
16. After STAB Variable2, what appears in Variable2? Why?
17. After STD Variable2, what appears in Variable2?
18. In the "Memory" window, locate \$2000. Highlight this byte, and you will see, in the top line of the window, its label. Do this for each of the first four bytes. What do you discover about the arrangement of the variables in memory?

19. Explain the bytes you see in Variable2 and the first byte of FailConst, based on the previously-executed command. What do you learn from this?
20. Explain what appears (or doesn't) in FailVar after `STAA FailVar`.
21. Explain what appears in A after `INX / LDAA 0,X`.

The previous exercise is extremely important in terms of your microcontroller programming future. When things go wrong, they will often do so because of a mishandling of memory defined for constants and variables. Bookmark this exercise! Here's a bit of a summary:

1. 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), and you can find a byte at a particular offset from that point.
2. 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`.
3. 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.
4. You can't load an address into an 8-bit register.
5. 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; if you load a 16-bit register, you will get two bytes: the one you pointed to and the one following it.
6. Storing a single byte into a multi-byte variable changes only the first byte.
7. Storing a 16-bit value to a single-byte variable will change the variable and the first byte of the next variable (affectionately known as clobbering the next variable).
8. Storing to a constant in ROM changes nothing – why else would we call it a "constant"?
9. Calling something in RAM "constant" is a lie – don't do it!
10. The Assembler lets you enter bytes into a constant or variable using a lot of different formats.