Lecture #6

Embedded Language Use

18-348 Embedded System Engineering

Philip Koopman

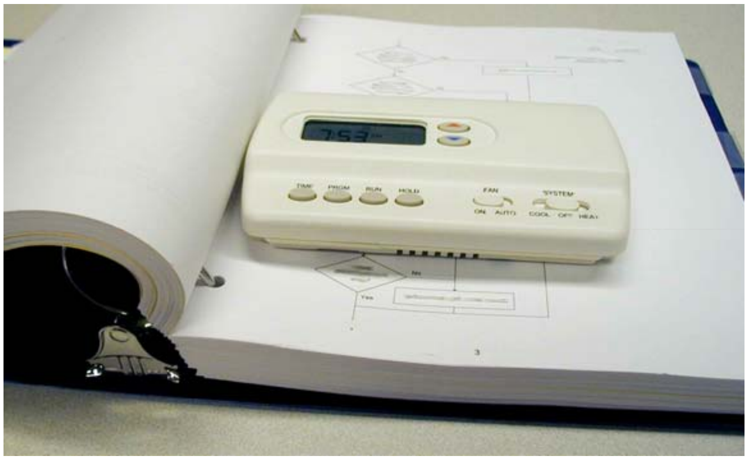
Monday, 1-Feb-2016

Electrical & Computer ENGINEERING

Carnegie Mellon

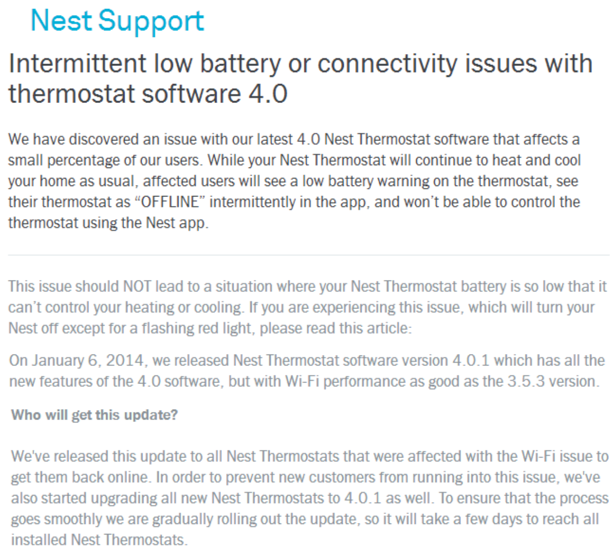
Household Thermostat (smarter than you think)

- **Household thermostat & flow charts**
  - Yes, they really are this complicated… easily 64KB program size
  - Example function: low battery warning
  - Newer generation is more connected (e.g., NEST learning thermostat)



NEST – A Smarter Thermostat

- **Version 4.0 released Nov 22nd 2013** (right before Thanksgiving Week)



Nest Support

Intermittent low battery or connectivity issues with thermostat software 4.0

We have discovered an issue with our latest 4.0 Nest Thermostat software that affects a small percentage of our users. While your Nest Thermostat will continue to heat and cool your home as usual, affected users will see a low battery warning on the thermostat, see their thermostat as "OFFLINE" intermittently in the app, and won't be able to control the thermostat using the Nest app.

This issue should NOT lead to a situation where your Nest Thermostat battery is so low that it can't control your heating or cooling. If you are experiencing this issue, which will turn your Nest off except for a flashing red light, please read this article:

On January 6, 2014, we released Nest Thermostat software version 4.0.1 which has all the new features of the 4.0 software, but with Wi-Fi performance as good as the 3.5.3 version.

Who will get this update?

We've released this update to all Nest Thermostats that were affected with the Wi-Fi issue to get them back online. In order to prevent new customers from running into this issue, we've also started upgrading all new Nest Thermostats to 4.0.1 as well. To ensure that the process goes smoothly we are gradually rolling out the update, so it will take a few days to reach all installed Nest Thermostats.

ARM Cortex A8 CPU
512Mb DRAM
2Gb NAND flash

http://www.ifixit.com/Teardown/Nest+Learning+Thermostat+2nd+Generation+Teardown/13818

Newer Thermostat Features

- **Internet connectivity**
  - Remotely set temperature
  - Send "I'll be home soon" command
  - Coordinate with lighting, occupancy, and multi-zone management

- **What could come next?** (many of these are already here…)
  - Automatically learn usage patterns
  - Coordinate temperature with weather
    - Start heating house in morning so it reaches desired temperature when you wake up
  - Coordinate temperature with power grid
    - Reduce cooling if power grid is overloaded
  - Heating/cooling diagnostics
    - Compare outside temperature to inside temperature and detect efficiency problems
  - Adaptive behaviors
    - Pre-cool when power is cheaper (use house mass as thermal capacitor)
  - Real-time smart grid management
    - Real-time energy auctions (set thermostat by $/day instead of temperature)

Where Are We Now?

- **Where we've been:**
  - Assembly language
  - Better engineering practices

- **Where we're going today:**
  - Embedded C
  - Embedded programming techniques

- **Where we're going next:**
  - More embedded programming techniques
  - Memory bus
  - Economics / general optimization
  - Debug & Test
  - Serial ports

  - Exam #1

Preview

- **Checklists**
  - Helping yourself get things right with minimum worry

- **Embedded-specific programming tricks**
  - Bit manipulation (C and Asm)
  - C keywords that matter for embedded

- **Combining C and Assembly programs**
  - How to link an assembly subroutine to a C calling program

Checklists

- **All people make mistakes**
  - Errors of commission (doing the wrong thing)
  - Errors of omission (forgetting to do something)
- **Errors of omission are more common – but easier to improve!**
  - Shopping lists
  - Post-it notes
  - Checklists (don't miss a step)
    - Packing for a trip
    - Performing a complex operation
    - Helping remember things in a crisis (e.g., aircraft emergency procedures)
    - Training new people
    - Helping experienced people who get bored and forget items

Course Checklists

- **Assignment checklist**
  - Each assignment has a checklist of items to hand in
- **Coding style sheet**
  - For every hand-in, go down the style sheet and make sure you got them right
- **Lab checklists**
  - Circuit debugging tips sheet – use it when you have a hardware problem
- **Maybe you want to make a hand-in checklist**
  - Check file name conventions
  - Load submissions back to your computer and make sure it compiles clean
  - … etc …
- **(Remember the design review checklist from last lecture? That is the sort of thing people in industry really use.)**

http://www.ece.cmu.edu/~ece348/labs/docs/lab_checklists.html

18-348 Lab Writeup Checklist

These checklists are to help you remember everything for pre-labs, lab demos, and lab writeups. It is your responsibility to read assignments carefully, and assignments have priority over this checklist. But, if you use this checklist, it will help you avoid simple mistakes. We encourage you to edit this checklist to make it more suitable for your own needs.

Pre-lab checklist:

- Come to recitation with all your prelab questions if you haven't completed the pre-lab by then.
- Complete all parts of pre-lab (use hand-in checklist included in pre-lab assignment)
- Ensure all files have course number, your group number, your andrew ID, and your name in them so they will be visible when printed.
- Follow the file naming convention: <base filename>_<andrewid>.<extension>
- Ensure all files other than source code are in Acrobat format and are legible when viewed.
- Transfer all files to designated afs space. Remember to use an ECE machine and not an Andrew machine for afs access.
- Transfer files back to your PC and check that (a) the files are in the correct subdirectory (including assignment number, assignment type, and early/ontime/late) (b) they are all there, and (c) they are all uncorrupted and legible. Using gzip or a similar program can help in detecting corruption, but it is important to unzip the files into the handin directory
- Hand in by 1:30 PM on Friday if possible to get bonus points.
- Hand in by 9:00 PM on Friday to get full credit.

Lab demo checklist:

- Before your demo time, make sure that everything in the demo works.
- Ensure that TA checks everything required on the demo list.
- Keep notes on everything you had to change to make the demo work if there were problems for your lab writeup.

Lab writeup checklist:

- Fix any problems with the lab noted by the TA
- Complete all parts of lab writeup (use hand-in checklist included in pre-lab assignment)
- Ensure all files have course number, your group number, your andrew ID, and your name in them so they will be visible when printed.
- Follow the file naming convention: <base filename>_g<group#>.<extension>
- Ensure all files other than source code are in Acrobat format and are legible when viewed.
- Transfer all files to designated afs space. Remember to use an ECE machine and not an Andrew machine for afs access.
- Transfer files back to your PC and check that (a) the files are in the correct subdirectory (including assignment number, assignment type, and early/ontime/late) (b) they are all there, and (c) they are all uncorrupted and legible.
- Hand in by 9:00 PM on Wednesday to get full credit.
- Remove all of your files from the lab computer you used.

Best Checklist Practices

- **Use the checklist!**
  - Emphasis originated in military organizations; NASA used them extensively
  - Just because you know the steps doesn't mean you will follow them!
- **Make your own local copy of any checklist**
  - Add things to the checklist if they bite you
  - Delete things from the checklist that aren't relevant
  - Keep the checklist length about right – only the stuff that really matters
- **Important uses of checklists in industry**
  - Testing – make sure you actually do all the tests
  - Design reviews – make sure you don't forget to look for things
  - Style sheets – make sure all style elements are met
- **A word about honesty**
  - Filling out a checklist when you didn't actually do the checks is dishonest!

Two's Complement Numbers (review)

Normal binary arithmetic on signed numbers

- **To take negative N-bit value, subtract from 2^N**
  - (8 bits) -13 = 256 – 13 = 243 = 0xF3

- **Limits on representation**
  - Maximum number is $(2^{N-1} - 1)$ 0x7F = 127 for 8 bits
    0x7FFF= 32767 for 16 bits
  - Minimum number is (-2^{N-1}) 0x80 = -128 for 8 bits
    0x8000= -32768 for 16 bits

- **Relevant assembly instruction:**
  - NEG, NEGA, NEGB

- **Ways to compute in C:**
  1. A = 0 – A; // do the subtraction
  2. A = (A ^ 0xFF) + 1; // invert all bits and add one

One's Complement Numbers

- **Some early computers did all computations in one's complement**
  - Still useful in some niche applications, particularly checksums

- **Limits on representation**
  - Maximum number is $(2^{N-1} - 1)$ 0x7F = 127 for 8 bits
    0x7FFF= 32767 for 16 bits
  - Minimum number is $(-(2^{N-1} - 1))$ 0x80 = -127 for 8 bits
    0x8000 = -32767 for 16 bits
  - Two values of zero: 0x00 and 0xFF 0x0000 and 0xFFFF

- **Relevant assembly instruction:**
  - COM, COMA, COMB
  - There is no built-in one's complement add – but there is a trick:
    take the carry-out and add it back in as the low bit (skips over 0x0000 value)

- **Ways to compute:**
  1. A = (A ^ 0xFF); // bit inversion is one's complement

Data Type Portability – what's an "int"?

- **In many embedded systems, "int" isn't 32 bits!**
  - And other things vary – char might default to signed or unsigned
- **Use more explicit data types to help with:**
  - Code portability (will run when moving from 8-bit to 16-bit CPU)
  - Your own portability (especially if you work with multiple CPUs)
  - Efficiency (so you don't drag big data types around on small CPUs)
- **Suggestions, with HC12-specific default definitions**
  - int8 – signed char / uint8 – unsigned char
  - int16 – signed int / uint16 – unsigned int
  - int32 – signed long / uint32 – unsigned long
  - int64 – (not supported) / uint64 – (not supported)

  Also consider:
  - bool – unsigned char
  - In Codewarrior, you can actually change lengths options, so be careful!
  - If you aren't sure, use "sizeof" in a test program to find out
    printf("Long Long is %d bytes\n", sizeof(long long));
- **Also see:**
  - **"stdint.h"** and **"inttypes.h"** for generic ways to deal with this issue
    - (The standard is e.g. "uint16_t" but "uint16" is used often in legacy systems)

Branch On Bits

- **Very often, embedded software deals with individual bits**
  - A few bits in a hardware status register that combines many bits
  - Packing data structures into bytes to save memory (e.g., 8 flags bits per byte)
- **Branch if bits clear – branch only if *ALL* of a set of bits are clear**
  - asm: BRCLR VARA, $A2, target_label
    branches if ***all*** of bits 1, 5, and 7 are clear (1010 0010)
  - C: if (!(VARA & 0xA2)) { … do if all bits clear… }
    executes "if" part if all bits clear (caution – reverse branch direction than asm)
- **Branch if bits set – branch only if *ALL* of a set of bits are set**
  - asm: BRSET VARA, $1C, target_label
    branches if ***all*** of bits 2, 3, and 4 are set (0001 1100)
  - C: if (!((VARA ^ 0xFF) & 0x1C)) { … do if all bits set … }
    executes "if" part if all bits set (caution – reverse branch direction than asm!)

Bit Setting and Clearing

- **Sometimes you need to set or clear a specific bit**
  - For example, to change values in a status register, or for packed data
  - Will work on multiple bits in a byte/word – examples are for only 1 bit

- **Setting a bit**
  - C: VARA = VARA | 0x40; // sets bit 6
  - asm: BSET VARA, $40

- **Clearing a bit**
  - C: VARA = VARA & (0x40 ^ 0xFF); // clears bit 6 (~0x40)
  - asm: BCLR VARA, $40

- **Inverting a bit**
  - C: VARA = VARA ^ 0x40; // inverts bit 6
  - (no single assembly instruction for direct addressing mode)

Bit Extraction

- **Sometimes multiple fields are packed into a single byte**
  - Example: color values in a “16-bit / 32K color” display word

1	5	5	5
X	RED	GREEN	BLUE

- **Here’s how to recover the fields from a single 16-bit RGB word:**
  - BLUE: blue_val = (RGB & 0x1F)
  - GREEN: green_val = ((RGB>>5) & 0x1F)
  - RED: red_val = ((RGB>>10) & 0x1F)

  - Note: better to mask after shift instead of before in case there is a signed/unsigned shifting problem

Bit Insertion

- **Inserting bits into a field is more difficult**
  - Clear out old bits in the field
  - OR in the new bits for the field

- BLUE: RGB = ((RGB & 0xFFE0) | ((new_blue & 0x1F)))
- GREEN: RGB = ((RGB & 0xFC1F) | ((new_green & 0x1F) << 5))
- RED: RGB = ((RGB & 0x83FF) | ((new_red & 0x1F) << 10))

Note: the “& 0x1F” is to prevent corruption from invalid value

- `1111 1111 1110 0000 = 0xFFE0`
- `1111 1100 0001 1111 = 0xFC1F`
- `1000 0011 1111 1111 = 0x83FF`

Bit Fields

```
struct mybitfield { flagA : 1;
                    flagB : 1;
                    nybbA : 4;
                    byteA : 8;
                  }
```

- **Bit Fields permit accessing memory in sections smaller than a byte**
  - Format: name : #bits; name : #bits; name : #bits;

```
struct mybitfield F;
F.flagA = 1;
F.flagB = 0;
F.nybbA = 0x0A;
F.byteA = 255;
F.flagB = F.flagB ^ 1;  // invert flag
```

- **Notes:**
  - Bit field order is undefined and up to the compiler, so don’t assume
  - Each bit field starts in a new machine word, unused bits are left idle in a word

C keyword: static

- **Static keyword for variables**
  - Tells compiler to put variable in a statically allocated piece of global memory
  - Variable is always in memory
  - Variable survives between calls
  - Alternative (non-static) is that variable is on stack

```
Void test_proc(void)
{ int8 a;
  static int8 b;
  printf("%d\n", a);   // undefined value from stack
  printf("%d\n", b);   // 17 on second and later calls
  b = 17;
}
```

- **Global != static**
  - "Global" has to do with scope – above is only visible within test_proc
  - "Static" has to do with it being a fixed memory address, not on the stack

The Other Use Of Keyword: "file static"

- **We hear that some companies ask this on job interviews…**
  - This is general C knowledge, and not particularly embedded-specific

- **When applied to a top level (global) variable *or* function definition**
  - Static means "not visible outside this compilation unit"
  - Or, more loosely, make this variable or function invisible outside a .c or .cpp
  - It works by omitting the information from the external linker map

```
static int8 b;
static void test_proc(void)
{ … }
```

- **Why would you want to do this?**
  - Most useful in C programs lacking ability to use C++ encapsulation features
  - Hides variable definition for encapsulation – other code can't link to it
  - Hides functions for encapsulation
    - In effect, functions inside the file are "friend" functions, and everything else isn't
  - Avoids naming collisions between globals and functions in multiple files

C Keyword: volatile

- **Volatile keyword for variables**
  - It means that something other than the C program can change its value…
    … so compiler doesn't keep it in a register (or cache) between computations
  - Must be loaded before each use and stored before each use
  - Memory-mapped I/O (e.g., reading from ports)
  - Interrupts that change values in background (e.g., reading time of day)

```
volatile bool lockout;
…..
// wait for lockout to be over
while (lockout) {/* do nothing */};
```

  - This is an infinite loop if lockout isn't volatile
  - It is just waiting for some other task to set the variable as written above

- **Globally visible variables may need to be volatile**
  - If two tasks share a variable, volatile makes sure changes are detected

Netrino

Bad Code in a Delay Loop

```
void Delay(uint16_t usec)
{
    uint16_t loopCount = 0;

    if (ScaleClock())
    {
        usec *= ((CLOCK_ENABLED +
                  (CLOCK_NORMAL – 1)) / CLOCK_NORMAL);
    }

    for (loopCount = 0; loopCount < usec; loopCount++)
    {
        /* wait */
    }
}
```

Not declared volatile. (points to line 3)

Loop may be optimized out!

C Keyword: inline

- **Inline keyword tells compiler to omit subroutine overhead**

```
inline int average( int a, int b)
{ int result;
  result = (a + b) / 2;
  return result;
}
```

- **These two generate identical code in most cases:**

```
c = average(a,b);
c = (a + b) / 2;
```

- **So why use it?**
  - Speeds up execution of very simple functions
  - Pasting the code instead of calling makes code more complex, less maintainable
  - A lot easier to do complex computations than a macro
  - Can remove “inline” keyword to save memory on multiple uses if desired
  - Similar to a #define macro, but cleaner and more scalable

Other Potential Keywords

- **ROM** – put this value in ROM (same idea as “const”)
- **RAM** – put this value in RAM
- **near / far** – designate as a 16-bit or larger pointer
- **large / small** – an array is bigger or smaller than 64 KB
- **interrupt** – a subroutine is an interrupt service routine and not a regular subroutine
- **register** – keep a value in a register instead of memory

Netrino

Unmaintainable Code

```
result = (d + 305) / 146097 * 400 + (d + 305) %
    146097 / 36524 * 100 + (d + 305) % 146097 % 36524 /
    1461 * 4 + (d + 305) % 146097 % 36524 % 1461 / 365;
```

What does this do? Is the operator precedence correct?

Why Macros Are Evil

- **You should use in-line rather than macros whenever possible**
  - Macros are the source of many nasty bugs
    - Almost impossible to debug without looking through expanded source code
  - "Whenever possible" is approximately equal to "ALWAYS"
  - If in-line function won't work, then you should use C++ templates

- **Unsafe macro:**

```
#define MYMACRO(x)    x%4
MYMACRO(a+b)      ➔       a+b%4
```

- **Macro rule – encapsulate every use of a macro parameter with "( )"**

```
#define MYMACRO(x)    (x)%4
MYMACRO(a+b)      ➔       (a+b)%4
```

Combining Assembly & C

- **Writing everything in assembly is usually a bad idea**
  - Too hard to get right
  - Too expensive – high level languages are less expensive to develop in
  - But, combining a little assembly with almost all C can be OK

1. **"In-line" assembly (only supported by some C compilers)**
   - This is very tricky to get right, and problems with changing registers

```
a = b + c;
__asm { …
      TFR   D,X
      LDY   2,SP
      ….
}
e = f + g;
```

2. **C program calls an assembly function**
   - This is what we will concentrate on in this course

Easiest Way To Start – Use An Example

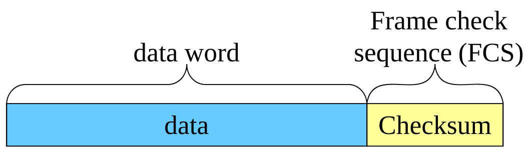
- **Best way to link to a C calling program is to use an example**
  - Create a dummy program with the same parameters you want to use
  - Run it through the compiler
  - Assembler version shows you how to access parameters
  - Modify assembler version to do what you want it to do
  - But, can be complex if optimizer mixes things up too much

- **Usually the best approach is:**
  - Create the code you want in C
  - See how bad the compiler is (maybe it isn't that bad after all!)
  - Tweak C code to try to help compiler out
  - Hand-tune assembly code only as a last resort

- **How do you actually get the C compiler to cough up assembly?**
  - -S option on most compilers generates assembly language output
  - In CW, can also use "disassemble" function

Compute A 1's Complement Checksum

- **Checksum – "Add" up all the bytes/words as a digital signature**
  - Reduces chance of data corruption going undetected



- **1's complement add is a lot better as a checksum than 2's complement!**
  - Gives 12.5% better error detection for 2-bit errors on 8 bit data

- **Design:**
  - Initialize a running sum (for example, to zero).
  - Loop across all bytes to be used in checksum. For each byte:
    - Add that byte to a running sum using two's complement (normal) addition
    - If there is a carry-out from the add, increment the running sum by one
      - » This skips over 0x00 in the addition, going from 0xFF to 0x01 if incrementing
  - Result left in running sum is the checksum

Program to do this:

Main:

```
static uint8 data[XLEN];
static uint8 cksum_result;
cksum_result = compute_ones_checksum(&data[0], 32);
```

Actual Routine:

```
uint8 compute_ones_checksum(uint8 *array, uint16 count)
{ uint8 *p;    uint16 checksum;   int i;
  checksum = 0;   p = array;
  for (i = 0; i < count; i++)  // add bytes 1 at a time
  { checksum = checksum + *(p++);  // add next byte
              // unsigned *p to avoid sign extension
    if (checksum & 0x100)        // check for carry-out
    { checksum++;                // wrap carry bit
      checksum = checksum & 0xFF;
 }  }
   return((uint8) checksum);}
```

Disassembled Procedure Call

```
// cksum_result = compute_ones_checksum(&x[0], 32);
  0015 cc0000     [2]     LDD    #x
  0018 3b         [2]     PSHD
  0019 c620       [1]     LDAB   #32
  001b 87         [1]     CLRA
  001c 0700       [4]     BSR    compute_ones_checksum
  001e 1b82       [2]     LEAS   2,SP    ; pop input
  0020 7b0000     [3]     STAB   cksum_result
```

- **Address passed on stack**
- **Last parameter passed in register D**
  - (A=0 B=32 => 16-bit "32") input count value
- **Result passed back out in B**

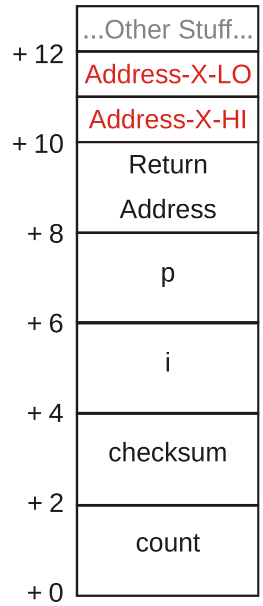
- **The "[1]" etc. are estimated cycle counts**

Disassembled Checksum Routine

```
  0000 6ca8   [2]     STD    8,-SP
//uint8 *p; uint16 checksum; int i;
//checksum = 0;
  0002 c7      [1]     CLRB
  0003 87      [1]     CLRA
  0004 6c82    [2]     STD    2,SP
// p = array;
  0006 ee8a    [3]     LDX    10,SP
  0008 6e86    [2]     STX    6,SP
// for (i = 0; i < count; i++)
  000a 6c84    [2]     STD    4,SP
  000c 201c    [3]   BRA    *+30
   ;abs=002a
// { checksum = checksum + *(p++);
  000e ee86    [3]     LDX    6,SP
  0010 e630    [3]     LDAB   1,X+
  0012 6e86    [2]     STX    6,SP
  0014 b714    [1]     SEX    B,D
  0016 e382    [3]     ADDD   2,SP
  0018 6c82    [2]     STD    2,SP
```

```
//  if (checksum & 0x100)
  001a 0f820107   [4]     BRCLR
   2,SP,#1,*+11 ;abs = 0025
// { checksum++;
  001e c30001 [2]     ADDD   #1
  0021 6c82    [2]     STD    2,SP
// checksum = checksum & 0xFF;
  0023 6982    [2]     CLR    2,SP
  0025 ee84    [3]     LDX    4,SP
  0027 08      [1]     INX
  0028 6e84    [2]     STX    4,SP
  002a ec84    [3]     LDD    4,SP
  002c ac80    [3]     CPD    0,SP
  002e 2dde [3/1]   BLT   *-32
   ;abs=000e
// } } return((uint8) checksum);
  0030 e683    [3]     LDAB   3,SP
  0032 1b88    [2]     LEAS   8,SP
  0034 3d      [5]     RTS
```

Stack Picture For That Code



Bet We Can Do Better Than That!

- **Things To Notice:**
  - C compiler wasn't very clever about keeping things in registers
    - Lots of accesses to the stack for storing intermediate values
  - Calling program has info needed to figure out where parameters are
    - Called program has them too, but can be very confusing!

- **Approach for this lecture**
  - Write a new called assembly language routine
  - Keep the same calling code – but replace subroutine code

  - (In real life you want to see if you can avoid assembly language altogether, but for this example we're assuming assembly language is the only practical choice)

Calling Program Framework

1. **Know where input parameters are**
   - Parameters pushed from left to right (first param deepest on stack)
   - ***Last parameter*** passed in a register
2. **Do the computation**
   - Use the stack for temporaries as needed
3. **Return result value**
   - ***Return result in register***

- **Complete info in CodeWarrior Assembler manual, Chapter 11 ("Mixed C and Assembler Applications")**

Size of Last Parameter	Type example	Register
1 byte	char	B
2 bytes	int, array	D
3 bytes	far data pointer	X(L), B(H)
4 bytes	long	D(L), X(H)

[Freescale]

Let's Rewrite The Example

Code Warrior Preliminaries:

- **Check both "C" and "assembler" options when building project**
  - Will create both a main.c and a main.asm
  - Put your assembly function in main.asm

- **Notes on main.c**
  - "int8 compute_ones_checksum(int8 *array, int count);"
    - That way compiler will know how to call it

- **Notes on main.asm**
  - Change asm_main to compute_ones_checksum
  - "XDEF compute_ones_checksum"
    - That way the linker will see it

Calling Parameters

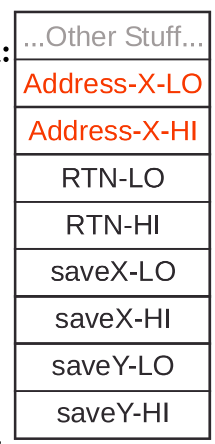
```
0015 cc0000     [2]     LDD   #x
0018 3b         [2]     PSHD
0019 c620       [1]     LDAB
 #32
001b 87         [1]     CLRA
001c 0700       [4]     BSR
 compute_ones_checksum
```

- D register has 16-bit count for input
- Return value in B   (8 bit result)

Get Parameters Into Registers

- **Strategy:**
  - B – is accumulated checksum
  - X – is pointer
  - Y – is loop counter

Stack:




```
compute_ones_checksum:
      PSHX                ; Save X & Y.
      PSHY
; D is input; not saved
      TFR     D,Y         ; Y register is count
      CLRA                ; D register is
      CLRB                ;   running checksum (init to 0)
      LDX     +6,SP       ; X register is array address
```

Perform A Checksum Loop Calculation

```
   INY      ; Increment Y and jump to decrement-and-test
   CLC      ; use repeated Add-with-carries for ones' complement
   BRA   TestDone   ;  this catches the case where count == 0

CKLoop:     ; Loop across array; Y is count, D value, X address
   ADCB  1,X+ ; add byte and post-increment X to point to next
              ; note: carry from one add goes into next add

TestDone:   DBNE    Y, CKLoop
   ADCB    #0    ; make sure last carry bit is accounted for
```

- **Notes:**
  - Increment Y and then decrement in test to catch count=0 case
    - Count is unsigned, so negative count isn't an issue
  - Repeated ADC does one's complement addition
    - (wraps 0xFF to 0x00)
  - INX walks X pointer across array while Y is being decremented

DBNE Decrement and Branch if Not Equal to Zero DBNE

Operation:

(Counter) – 1 ⇒ Counter
If (Counter) not = 0, then (PC) + $0003 + Rel ⇒ PC

Description:

Subtract one from the specified counter register A, B, D, X, Y, or SP. If the counter register has not been decremented to zero, execute a branch to the specified relative destination. The DBNE instruction is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

IBNE and TBNE instructions are similar to DBNE except that the counter is incremented or tested rather than being decremented. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
DBNE *abdxys*, *rel9*	REL	04 lb rr	PPP/PPO	PPP

1. Encoding for lb is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (DBEQ – 0) or not zero (DBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 would be 0:0 for DBNE.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	DBNE A, *rel9*	04 20 rr	04 30 rr
B	001	DBNE B, *rel9*	04 21 rr	04 31 rr
D	100	DBNE D, *rel9*	04 24 rr	04 34 rr
X	101	DBNE X, *rel9*	04 25 rr	04 35 rr
Y	110	DBNE Y, *rel9*	04 26 rr	04 36 rr
SP	111	DBNE SP, *rel9*	04 27 rr	04 37 rr

[Freescale]

Clean Up And Return Results

```
; low 8 bits of result is already in B register; return value

   PULY    ; restore saved X & Y
   PULX
   RTS                    ; return to caller
```

- **Notes:**
  - We were careful to use B for the calculation, so just pass back in B
  - Restore X & Y in case being used by main

Complete Optimized Assembly Version

```
compute_ones_checksum:
            PSHX
            PSHY
            TFR     D,Y
            CLRA
            CLRB
            LDX     +6,SP
            INY
            CLC
            BRA     TestDone
CKLoop:     ADCB    1,X+
TestDone:   DBNE    Y, CKLoop
            ADCB    #0
            PULY
            PULX
            RTS
```

- **Notes:**
  - This is a lot smaller and faster than the C compiler!
    - 2 instructions in the inner loop
    - Problem – C language can't represent a carry bit
    - Problem – C compilers not always smart about loops & register use
  - We'll look at such things further in the optimization lectures

Review

- **Checklists**
  - Actually use the checklists we're giving you (and update them for yourselves)
- **Embedded-specific programming tricks**
  - Bit manipulation (C and Asm)
    - Set, clear, invert, extract, insert bits
  - C keywords that matter for embedded
    - What the keywords do: static, volatile, inline
- **Combining C and Assembly programs**
  - How to link an assembly subroutine to a C calling program
    - Value passing (last value in register; rest on stack)
    - How all the pieces fit together
  - I don't expect you to be super-amazing at crazy optimization on a test

Lab Skills

- **Use bit manipulation instructions**
  - Move bits around
  - Insert/extract bits with and without C compiler bit fields
- **Combine C and Assembly routines**
  - Create an assembly language routine called by a C program