EE 319K
Introduction to Embedded Systems

Lecture 5: Pointers, Indexed Addressing Mode, Functional Debugging, Arrays, Strings, Timer

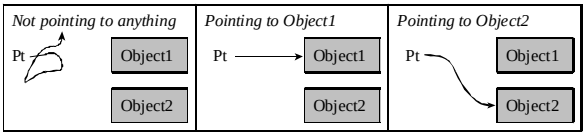
Announcements

- Homework
  - No homework next Monday
  - Next homework (Homework 4) 2/27
- Lab 3
  - Next Tuesday/Wednesday
  - Switch and LED interfacing
    - Implement Lab 2 on board
- Exam 1
  - Next Thursday, Feb. 23, in class (2-3:30pm)
    - Closed book, closed notes
  - Review next Tuesday

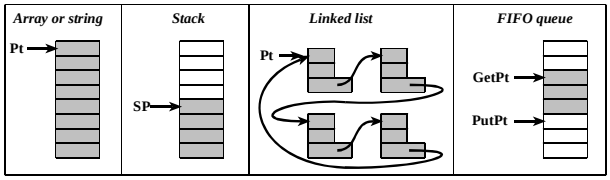
Agenda

- Recap
  - Board intro
    - Switch and LED interfacing
    - Heartbeat for debugging
  - Refinement
    - Control structures: if-then-else, loops
  - Modular design
    - Subroutines & parameters
- Outline
  - Pointers and indexed addressing
  - Functional debugging
  - Timers

Pointers

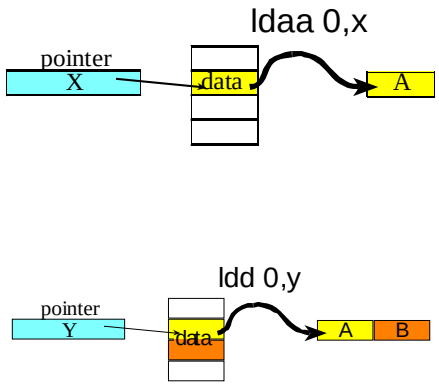


- Pointers are addresses that refer to objects
  - Objects may be data, functions or other pointers



  - If register X or Y contains an address we say it points into memory

Use of Index registers



Indexed Addressing Mode

Indexed addressing uses a fixed offset with the 16-bit registers: X, Y, SP, or PC. It takes the following 3 forms:

Instruction	Object / Machine Code
op oprx0_xysp	**69 xb**
op oprx9, xysp	**69 xb ff**
op oprx16, xysp	**69 xb ee ff**

Ramesh Yerraballi

5-6

$69 is the opcode for the `clr` instruction. Op could be one of several instructions that support the indexed addressing mode, e.g., adda, subd, jsr,

oprx0_xysp

Operand	Meaning
oprx3,–xys	Predecrement X or Y or SP by 1 . . . 8
oprx3,+xys	Preincrement X or Y or SP by 1 . . . 8
oprx3,xys–	Postdecrement X or Y or SP by 1 . . . 8
oprx3,xys+	Postincrement X or Y or SP by 1 . . . 8
oprx5,xysp	5-bit constant offset from X or Y or SP or PC
abd,xysp	Accumulator A or B or D offset from X or Y or SP or PC

Examples - Table 1 of the RG

clr A, X

- Opcode is $69
- clear the memory location [(X)+(A)]
- Operand can be figured out from Table 1 as $E4
- Object Code: $69E4

clr -2, Y

- Opcode is $69
- clear the memory location [(Y)-2]
- Operand can be figured out from Table 1 as $5E
- Object Code: $695E

clr 1,Y+

- Opcode is $69
- clear [(Y)] and Increment Y by 1
- Operand can be figured out from Table 1 as $70
- Object Code: $6970

9-bit and 16-bit - Table 2 of RG

Machine Code: **69 xb ff** (9-bit);
69 xb ee ff (16-bit)

$xb: Represents the bit pattern **%111rr0zs**

%rr: Which register (X:00,Y:01,SP:10,PC:11)

%z: 9-bit (0) or 16-bit (1) offset

%s: Sign bit of the offset

$ff: Represents the lower 8 bits of the offset

$ee: Represents the higher byte of the 16-bit offset

Example: **clr 150, Y**

$xb: %111 01 0 0 0; $ff: %10010110

Object Code: **$69 E8 96**

Assembly vs. C

```
Prime fdb
    1,2,3,5,7,11,13,17,19,23
```

1. Want to fetch the 7 from **`Prime[4]`**

```
ldx #Prime ;pointer to structure
ldd 8,x    ;read element num 4
```

- Or we could have fetched it directly as

```
ldd Prime+8  ;read Prime[4]
```

```
unsigned short const
  Prime[10] = {
  1,2,3,5,7,11,13,17,19,23
};
```

Assembly vs. C

```
Prime fdb 1,2,3,5,7,11,13,17,19,23
```

2. Want to increment the pointer to the next element.

```
Pt  rmb 2 ;16-bit pointer to Prime
```

and initialize it as

```
ldx #Prime
stx Pt   ;pointer to Prime[0]
```

to increment the pointer to the next element we have to add 2 (double byte) to the pointer.

```
ldx Pt  ;previous pointer
inx
inx    ;next element in the 16-bit
       ; structure
stx Pt
```

```
unsigned short const Prime[10]=
{1,2,3,5,7,11,13,17,19,23};
```

2. Want to increment the pointer to the next element.

we define the pointer as

```
unsigned short const *Pt;
```

and initialize it as

```
Pt =  Prime;
```

increment the pointer to the next element

```
Pt++;
```

Indexed Addr. Mode: Variants

Post-increment addressing first accesses the data then adds to the index register:

```
staa 1,Y+ ; Store at 2345,
          ; New Reg Y=2346
staa 4,Y+ ;Store at 2345,
          ; then Reg Y=2349
```

Pre-increment addressing first adds to the index register then accesses the data:

```
staa 1,+Y ;Reg Y=2346,
          ; then store at 2346
staa 4,+Y ;Reg Y=2349,
          ; then store at 2349
```

Post-decrement addressing first accesses data then subtracts from index register:

```
staa 1,Y- ;Store at 2345,
          ; then Reg Y=2344
staa 4,Y- ;Store at 2345,
          ; then Reg Y=2341
```

Pre-decrement addressing first subtracts from index register then accesses the data:

```
staa 1,-Y ;Reg Y=2344,
          ; then storeat 2344
staa 4,-Y ;Reg Y=2341,
          ; then store at 2341
```


Assume RegY has 2345 in it

Indexed Addr. Mode: Variants

Accumulator Offset Indexed addressing mode

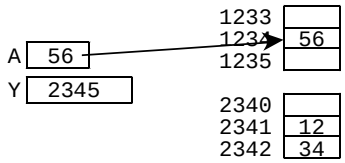
The offset is located in one of the accumulators A, B or D, and the base address is in one of the 16-bit registers: X, Y, SP, or PC.

```
ldab #4
ldy  #2345
staa B,Y  ;Store at 2349
          ;(B & Y unchanged
```

- *Accumulator offset indexed addressing is efficient for accessing arrays. We can place the index in an accumulator and the base address in Reg X or Reg Y*

Indexed Indirect addressing mode

```
ldy  #2345
staa [-4,Y] ; fetch 16-bit address
            ; from 2341, which is
            ; i.e., 1234
            ; store 56 at 1234
```



Load Effective Address

```
leax idx ;RegX=EA
leay idx ;RegY=EA

leas idx ;RegS=EA
```

Examples

```
leax m,r       ;IDX 5-bit index, X=r+m (-16 to 15)
leax v,+r      ;IDX pre-inc, r=r+v, X=r  (1 to 8)
leax v,-r      ;IDX pre-dec, r=r-v, X=r  (1 to 8)
leax v,r+      ;IDX post-inc, X=r, r=r+v  (1 to 8)
leax v,r-      ;IDX post-dec, X=r, r=r-v  (1 to 8)
leax A,r       ;IDX Reg A offset, X=r+A, zero padded
leax B,r       ;IDX Reg B offset, X=r+B, zero padded
leax D,r       ;IDX Reg D offset, X=r+D
leax q,r       ;IDX1 9-bit index, X=r+q (-256 to 255)
leax W,r       ;IDX2 16-bit index, X=r+W (-32768 to 65535)
```

Where `r` is Reg X, Y, SP, or PC, and the fixed constants are:
`m` is any signed 5-bit value (-16 to +15); `q` is any signed 9-bit value (-256 to +255); `v` is any unsigned 3 bit value (1 to 8); `W` is any signed 16-bit value (-32768 to +32767 or any unsigned 16-bit 0 to 65535)

Functional Debugging

Instrumentation: dump into array without filtering
Assume **happy** is strategic 8-bit variable.

`SIZE  equ 20`
`Buf  rmb SIZE`
`Pt  rmb 2`
	`#define SIZE 20`
	`unsigned char Buf[SIZE];`
	`unsigned char *Pt;`

Pt will point into the buffer. **Pt** must be initialized to point to the beginning, before the debugging begins.	
`ldx #Buf` `stx Pt`	`Pt = Buf;`

... Functional Debugging

The debugging instrument saves the strategic variable into the **Buffer**.

```
Save
  pshb
  pshx    ;save
  ldx  Pt ;X=>Buf
  cpx  #Buf+SIZE
  bhs  done ;skip if full
  ldab happy
  stab 0,X ;save happy
  inx      ;next address
  stx  Pt
done
  pulx
  pulb
  rts
```

```
void Save(void){
  if(Pt < &Buf[SIZE]){
    (*Pt) = happy;
    Pt++;
  }
}
```

Next, you add **jsr Save** statements at strategic places within the system.
Use the debugger to display the results after program is done

Arrays

- **Random access**
- **Sequential access**.
- An **array**
  - o equal precision and
  - o allows random access.
- The **precision** is the size of each element.
- The **length** is the number of elements (fixed or variable).
- The **origin** is the index of the first element.
- **zero-origin indexing**.

... Arrays

- In general, let **n** be the precision of a zero-origin indexed array in bytes.
- If **I** is the index and
- **Base** is the base address of the array,
- then the address of the element at **I** is

  Base+n*I

- Length of the Array:
  - One simple mechanism saves the length of the array as the first element.

C:

```
const char Data[5] = {
  4,0x05,0x06,0x0A,0x09
};
const short Powers[6] = {
  5,1,10,100,1000,10000
};
```

Assembly:

```
Data fcb 4,$05,$06,$0A,$09
Powers fdb 5,1,10,100,1000,10000
```


With the keyword `const` the C compiler forces the array to be placed in ROM.

... Arrays

- Length of the Array:
  - Alternative mechanism stores a special termination code as the last element.

ASCII	code	Name
NUL	$00	null
ETX	$03	end of text
EOT	$04	end of transmission
FF	$0C	form feed
CR	$0D	carriage return
ETB	$17	end of t ransmission block

Example 6.1

Statement: *Write a software module to control the read/write (R/W) head of an audio tape recorder. From the perspective shown in Figure 6.8, the stepper motor causes the R/W head to move up and down. This motion affects which audio track on the tape is under the head. The goal is to be able to move the motor one step at a time.*

Solution: This module requires three public function: one for initialization, one to rotate one step clockwise, and one to rotate one step counter-clockwise. By rotating the motor one step at a time, the software can control which audio track on the tape is under the R/W head. A stepper motor has four digital control lines. To make the stepper motor spin, we output the sequence 5,6,10,9 over and over on these four lines. To make it spin in the other direction, we output the sequence in the other direction. This motor has 24 steps per revolution, therefore one step will change the shaft angle by exactly 15º. To make the motor step once, we output just the next number in the sequence. For example, if the output is currently at 5, and we wish to rotate the shaft by 15º, we simply output a 6. In this solution, we will store the 5,6,10,9 data in an array, as shown in Figure below:

$4000	$05
$4001	$06
$4002	$0A
$4003	$09

...Solution

```
org  $0800  ;RAM
Index rmb  1   ;0,1,2,3
      org  $4000  ;ROM
Data  fcb $05,$06,$0A,$09
Stepper_Init
      bset DDRT,#$0F
      movb #$09,PTT
      clr  Index
      rts
Stepper_CW
      ldab Index
      ldx  #Data
      ldaa B,X
      staa PTT  ;15 deg
      incb      ;CW
      andb #$03
      stab Index
      rts
```

```
unsigned char Index;

const char Data[4]=
    {0x05,0x06,0x0A,0x09};

void Stepper_Init(void){
  DDRT |= 0x0F; // PT3-0 are outputs
  PTT = 0x09;  // first data
  Index = 0;  // first index
}
void Stepper_CW(void){
  PTT = Data[Index]; // rotate 24deg
  Index = 0x03&(Index+1); // next index
}
```

Program 6.1

Strings

A **string** is a data structure with

- ❖ equal size elements that only allows sequential access.

Problem: *Write software to output a sequence of values to a digital to analog converter.*

Solution: In this system, the length of the string is stored in the first byte. This approach is appropriate when the data elements can take on all possible numeric values. Assume a DAC converter is connected to Port T. The function **DAC**, shown in the next slide, will output the string data to the DAC. The function uses ***call by reference***, meaning a pointer to the data is passed. The main program calls this function twice, with different data strings.

...Solution

```
Data1 fcb  4            ;length
      fcb  0,50,100,50  ;data
Data2 fcb  8
      fcb  0,25,50,75,100,75,50,25
*Reg X points to the string data
DAC   ldab 0,x   ;length
loop  inx        ;next element
      ldaa 0,x   ;data
      staa PTT   ;out to DAC
      decb
      bne  loop
      rts
main  lds  #$4000
      movb #$FF,DDRT
mloop ldx  #Data1  ;first string
      bsr  DAC
      ldx  #Data2  ;second string
      bsr  DAC
      bra  mloop
```

```
unsigned const char Data1[5]=
  {4,0,50,100,50};
unsigned const char Data2[9]=
  {8,0,25,50,75,100,75,50,25};
void DAC(unsigned char *pt){
  unsigned int length;
  length = (*pt++);  // size
  do{
    PTT = (*pt++);
  }
  while(--length);
}
void main(void){
  DDRT = 0xFF;  // outputs to DAC
  while(1){
    DAC(Data1); // first string
    DAC(Data2); // second string
  }
}
```

Program 6.5

Strings

Problem: *Write software to output an ASCII string to the serial port.*

Solution: Because the length of the string may be too long to place all the ASCII character into the registers at the same time, ***call by reference*** parameter passing will be used. With call by reference, a pointer to the string will be passed. The function **`OutString`**, will output the string data to the serial port.

The function **`SCI_OutChar`** will be developed later in Chapter 8 and shown as Program 8.2. For now all we need to know is that it outputs a single ASCII character to the serial port. The main program calls this function twice, with different ASCII strings.

...Solution

```
Hello fcc  "Hello World"
      fcb  0
CRLF  fcb  13,10,0
;Reg X points to the string data
OutString
      ldaa 1,x+    ;next data
      beq  done    ;0 means end
      jsr  SCI_OutChar
      bra  OutString
done  rts
main  lds  #$4000
      bsr  SCI_Init
mloop ldx  #Hello  ;first string
      bsr  OutString
      ldx  #CRLF   ;second string
      bsr  OutString
      bra  mloop
```

```
unsigned const char CRLF[3]=
  {13,10,0};

void OutString(unsigned char *pt){
  unsigned char letter;
  while(letter = (*pt++)){
    SCI_OutChar(letter);
  }
}
void main(void){
  SCI_Init();
  while(1){
    OutString("Hello World");
    OutString(CRLF);
  }
}
```

Program 6.6

Timer

9S12 timer ports.

Addr	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0044	**Bit 15**	**14**	**13**	**12**	**11**	**10**	**9**	**Bit 8**	**TCNT**
$0045	**Bit 7**	**6**	**5**	**4**	**3**	**2**	**1**	**Bit 0**	**TCNT**
$0046	**TEN**	TSWAI	TSFRZ	TFFCA	0	0	0	0	TSCR1
$004D	TOI	0	0	0	TCRE	PR2	PR1	PR0	TSCR2
$004F	TOF	0	0	0	0	0	0	0	TFLG2

				E = 8 MHz		E = 24 MHz	
PR2	PR1	PR0	Divide by	TCNT period	TCNT frequency	TCNT period	TCNT frequency
0	0	0	1	125 ns	8 MHz	41.7 ns	24 MHz
0	0	1	2	250 ns	4 MHz	83.3 ns	12 MHz
0	1	0	4	500 ns	2 MHz	167 ns	6 MHz
0	1	1	8	1 µs	1 MHz	333 ns	3 MHz
1	0	0	16	2 µs	500 kHz	667 ns	1.5 MHz
1	0	1	32	4 µs	250 kHz	1.33 µs	750 kHz
1	1	0	64	8 µs	125 kHz	2.67 µs	375 kHz
1	1	1	128	16 µs	62.5 kHz	5.33 µs	187.5kHz

Given an E clock frequency, the PR2 PR1 and PR0 bits define the TCNT rate.

* The highlighted values for TCNT frequency in the bottom left corner of the table are wrong in the book

Ramesh Yerraballi

5-26

The Timer using TCNT is a programmable counter. That is, imagine a counter (16-bit number) that counts up from 0 to 2^{16}-1 (65535) and then overflows and goes to 0 and repeats this process. So, once the TCNT counter is *enabled*, it begins its counting up. The memory locations pertinent to the TCNT timer are at locations $0044,$0045, $0046, $004D and $004F. We refer to them as TCNT registers with symbolic names TCNT($0044,$0045), TSCR1($0046), TSCR2($004D) and TOF($004F). We will see the use of TOF flag later when we cover interrupts.

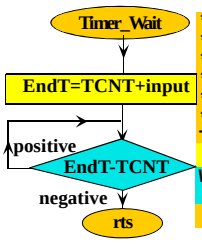
The 16-bit counter is the TCNT register which contains the current value of the counter in its count up. We have to set the TEN bit (bit 7 of TSCR1) to enable the counting. The programmable nature of the counter comes into play with the 3 bits PR2,PR1 and PR1 of the TSCR2 register, referred to as the Prescale value of the timer. The *Prescale* value, together with the E-clock frequency (which you recall is 8 MHz or 24 MHz depending on whether you are in run or load mode) determines the rate at which the counter ticks. Specifically, the tick (one count is one tick) rate is E-clock/2^{prescale}. Here are some examples:

- If E-clock is 8 MHz and the *Prescale* is set to 0 (PR2=0, PR1=0 and PR0=0; $(000)_2$) then the count rate is $8/2^0$ = 8 MHz, which gives a time delay between ticks of 125ns.
- If E-clock is 24 MHz and Prescale is set to 5 (PR2=1,PR1=0 and PR0=1; $(101)_2$) then the count rate is $24/2^5$ = 750 KHz, which gives a time delay between ticks of 1.33us.

Higher the *Prescale* value lower the tick-rate and therefore higher the delay between ticks. In other words, you cannot make the counter tick faster than the clock speed, you can slow it down though.

Fixed Time Delay

```
Timer_Init
        movb #$80,TSCR1   enable TCNT
        movb #$00,TSCR2  ;125ns in RUN
        rts
```




```
****Timer_Wait**********
* Time delay function
* Input: RegD is the time to wait
* (in 125ns cycles)
* Outputs: none
* error: input must < 32767
Timer_Wait
        addd TCNT     value at end
Wloop cpd  TCNT     EndT<Tcnt?
        bpl  Wloop
        rts
```

Ramesh Yerraballi 5-27

So, how do we use this counter to write a delay routine?

The idea is similar to the delay routine you used in Lab2 where you executed some dummy idiv instructions in a loop. However, here we do not execute any dummy instructions but repeatedly check to see if the TCNT has reached a target count that tells us that the desired amount of time has expired. For example, if we wanted a delay of half a second (500ms) then we can program the counter to tick with say a delay between ticks of 500us and set a target count of a thousand of these ticks, which gives us 500ms. The `Timer_Init` subroutine on this slide shows how to enable the Timer (`TEN` bit is set to 1) and the set the Prescale value to 0, which, for a E-clock frequency of 8 MHz (RUN mode) sets the tick delay to 125ns. Here is a little math shortcut:

> *With E-clock at 8 MHz we know that the clock cycle is 125ns. So, when the Prescale value is 2, the tick delay is $125*2^2 = 125*4 = 500$. The idea of this shortcut is to avoid having to do any divisions. Similarly, if E-clock is 24 MHz then the clock cycle is 41.7ns and with a Prescale value of say 5, the tick delay is $41.7*2^5=41.7*32 = 1333$ns.*

The `Timer_Wait` subroutine takes a parameter (count) in register D and then checks when that many ticks of the counter have elapsed before returning. Therefore, effectively delaying for a time equal to the the tick delay (set by the init subroutine) times the passed count. It achieves this task by setting the target count to be whatever the current value of TCNT is, plus the desired count (addd does this). Note that the TCNT could be any value, because the counter started when the TEN bit was set. Now, it repeatedly checks to see if the TCNT has reached this target count:

> The cpd instruction compares the target time (in D) against the current value of TCNT (which you recall changes continuously). The bpl (branch if plus) statement checks to see if the comparison results in a positive value indicating that the target time is not reached yet, in which case it goes back and re-performs the check.

Once the target count is reached the bpl instruction does not branch and the subroutine returns.

Example 1

- Assume
  - TCNT = 31 (can be any value)
  - Reg D = 4000 (means 0.5ms, in 125-ns units)
  - The **addd** will make RegD = 4031 (remains fixed for the rest of the subroutine)
- Run **square.rtf**, with a ScanPoint on **tloop**

```
                                          RegD-TCNT
CCR=sXhInzvc RegD=4031 TCNT=34     3997
CCR=sXhInzvc RegD=4031 TCNT=40     3991
CCR=sXhInzvc RegD=4031 TCNT=46     3985
CCR=sXhInzvc RegD=4031 TCNT=52     3979
…
CCR=sXhInzvc RegD=4031 TCNT=4000     31
CCR=sXhInzvc RegD=4031 TCNT=4006     25
CCR=sXhInzvc RegD=4031 TCNT=4012     19
CCR=sXhInzvc RegD=4031 TCNT=4018     13
CCR=sXhInzvc RegD=4031 TCNT=4024      7
CCR=sXhInzvc RegD=4031 TCNT=4030      1
CCR=sXhINzvC RegD=4031 TCNT=4036     -5
```

Ramesh Yerraballi 5-28

Say we set a scanpoint (in TExaS) on line with the label tloop (of Timer_Wait), we would see the TheLog window show the values of the watch variables listed in the Viewbox. Here, we are watching the condition code register, the register D and the memory location TCNT. Note, that the Square.rtf code on Jon's site sets the Prescale to 5 and the count to 10000 and not 0 and 4000 respectively to create a square wave of a 80 ms period. So you will not get the same values as shown here unless you modify it.

The purpose of the above example is to show how the value of register D is compared against the ticking count of TCNT till the N bit is set (bpl) to indicate that TCNT has gone past the value in register D.

In example 1, we are assuming that the TCNT value when the delay routine is called is 31, in example 2 on the next slide we see that even if the TCNT value were large (64442), the technique still works. As, the target time is set to 64442+4000 which causes an overflow with the resulting value being 2903 (68442 - 65536 = 2903). The cpd instruction performs RegD-TCNT which starts out at 2903-64442 which is a journey of 64442 steps in the counter-clockwise direction of the 16-bit number wheel. This puts us at 3997:

Successive checks leave the N bit 0 till TCNT value passes 2903.

Example 2

- Assume
  - TCNT = 64439 (can be any value),
  - Reg D = 4000 (means 0.5ms, in 125-ns units)
  - The **addd** will make RegD = 2903 (remains fixed for the rest of the subroutine)
- Run **square.rtf**, with a ScanPoint on **tloop**

```
                                          RegD-TCNT
CCR=sXhInzvc RegD=2903 TCNT=64442  3997
CCR=sXhInzvc RegD=2903 TCNT=64448  3991
CCR=sXhInzvc RegD=2903 TCNT=64454  3985
…
CCR=sXhInzvc RegD=2903 TCNT=65528  2911
CCR=sXhInzvc RegD=2903 TCNT=65534  2905
CCR=sXhInzvc RegD=2903 TCNT=4      2899
CCR=sXhInzvc RegD=2903 TCNT=10     2893
…
CCR=sXhInzvc RegD=2903 TCNT=2896   7
CCR=sXhInzvc RegD=2903 TCNT=2902   1
CCR=sXhINzvC RegD=2903 TCNT=2908   -5
```

Ramesh Yerraballi 5-29

As the note on the `Timer_wait` subroutine says, there is a restriction on the count value that it has to be less than 32767. Why?

Consider the case when running into the loop the first time, what is essentially computed is:

yellow: D = D_input + TCNT

blue: while ((D – TCNT) >= 0) /* do nothing */;

Now, assume that TCNT has not moved between the initialization (yellow) and the first loop check (blue). What the loop then essentially checks is whether

(D_input + TCNT – TCNT) >= 0 (on the number wheel: first clockwise, then counterclockwise TCNT steps)

So it loops only if D_input is positive. If D_input > 327676, the most-significant bit (MSB), bit 15, is set, i.e. it represents a negative number and the loop immediately exits. The root cause of the error is the mixing of an unsigned input with a signed comparison operation (bpl). To be consistent, we need to treat the input as a signed number, where anything greater than 32767 is a negative input value, which won't work as intended.

So, the highest delay we can achieve with this wait technique is limited to :

Highest delay is with the slowest clock rate(8 MHz) and highest Prescale (8) and the maximum count (32767). Which gives us:

tick delay = $125*2^8$ = 16us

and

Total Timer_wait delay = 32767*16us = 0.52 secs (half a second).

So, how can we delay for more than half a second? Well, we could call the `Timer_wait` routine several times to accomplish this. A more elegant solution is to write a Timer wait routine that waits for say 1 ms and then call it 1000 times to create a 1 second delay and so on. This is what you are being asked to do in Lab4.

Fixed Time Delay: C Version

```
// 9S12DP512 at 8 MHz; Init TCNT to 1us
// Input: none; Output:none
void Time_Init(void) {
   TSCR1 = 0x80; // Enable TCNT
   TSCR2 = 0x03; // Divide by 8 => 1us per count
}
void Timer_Wait(unsigned short cycles){
   unsigned short startTime = TCNT;
   while ((TCNT-startTime) <= cycles){}
}
// Input: Delay time in 1ms units
// Output: None
void Timer_Wait1ms(unsigned short delay){
   unsigned short i;
   for(i=0; i<delay; i++){
       Time_Wait(1000); //wait 1ms
   }
}
```


The proper way to accept the full range of unsigned inputs, is to do all computations and comparisons in the unsigned domain.

This is such a C version, which does not have the restriction of 32767. It can go up to 65000.

Assembly Mimicking C

```
;**** Timer_Wait ****
; Fixed Time Delay Function like C
; Input: RegD has time to Wait (125ns units)
Timer_Wait
       std    delay         ; time to wait
       movw   TCNT,start    ; TCNT at start
Wloop  ldd    TCNT          ; TCNT now
       subd   start         ; soFar = TCNT-startcpd
       cpd    delay         ;
Wchk   blo    wloop         ; soFar <= delay
       rts
```

This version works like the C version, so it relaxes the restriction on the delay up to 65000.

The same in assembly, it does not have the restriction of 32767 and can go up to 65000.

But: More complex code, more instructions, more overhead.

Real-Time Systems

- Bounded Latency
  - Input interface:
    input interface latency RDRF -> Read SCIDRL
  - Output interface:
    output interface latency TDRE -> Write SCIDRL
  - Periodic process
    (square wave, Labs 6, 7 and 8):
    - Software activity occurs at a periodic rate, Δt
    - Let t_n be the nth time the process executes
    - Goal is to make $t_n - t_{n-1} = \Delta t$

      δt=jitter $\Delta t - \delta t < t_i - t_{i-1} < \Delta t + \delta t$ for all i