Recap

Subroutines, the stack, switches, LEDs

Overview

Pointers
Indexed mode addressing
TCNT (free running 16-bit time)
Introduction to Lab 3
Can we collect data to prove it works?
Input, output, time

Read sections 4.5, 6.1, 6.2, 6.3 and 6.11

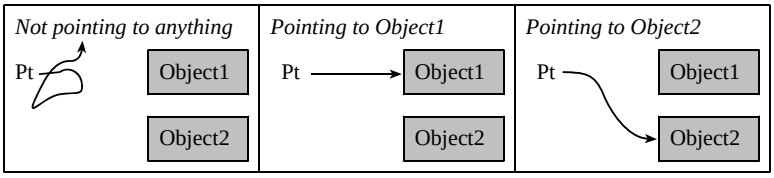


Figure 6.1. Pointers are addresses pointing to objects. The objects may be data, functions, or other pointers.

6.1. Indexed addressing modes used in implement pointers

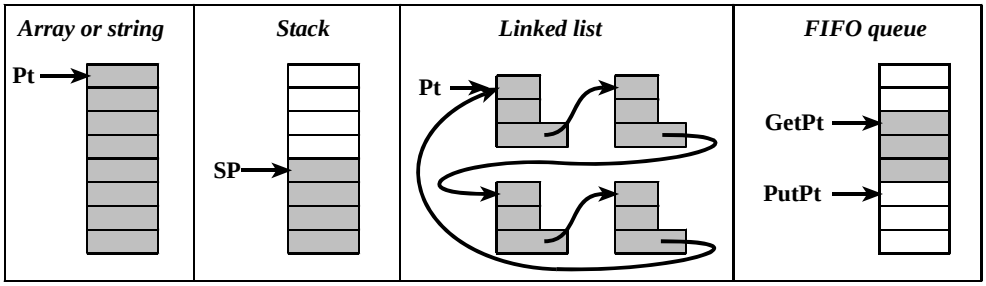


Figure 6.2. Examples of data structures that utilize pointers.

If Register X or Y contains an address, we say it points into memory

Definitions in C

```
unsigned char data;   // 8-bit value
unsigned char out;    // 8-bit value
unsigned char *pt;    // 16-bit address
```

Definitions in assembly

```
data rmb 1    ; 8-bit value
out  rmb 1    ; 8-bit value
pt   rmb 2    ; 16-bit address
```

Initialization in C

```
  pt = &data;  // pointer to data
```

Initialization in assembly

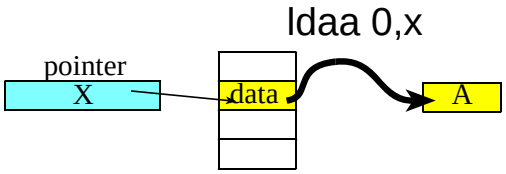
```
  ldx #data    // pointer to data
  stx pt
```

Dereference in C

```
   out = *pt;    // fetch value at pointer
```

Dereference in assembly

```
   ldx  pt   ; X points to data
   ldaa 0,x  ; fetch value at pointer
   staa out
;read 8-bit contents pointed to by X
```



16-bit definitions in C

```
unsigned short data;   // 16-bit value
unsigned short out;    // 16-bit value
unsigned short *pt;    // 16-bit address
```

16-bit definitions in assembly

```
data rmb 2    ; 16-bit value
out  rmb 2    ; 16-bit value
pt   rmb 2    ; 16-bit address
```

Initialization in C

```
  pt = &data;  // pointer to data
```

Initialization in assembly

```
  ldx #data    // pointer to data
  stx pt
```

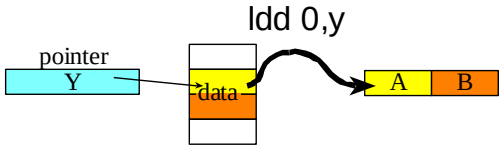
16-bit dereference in C

```
   out = *pt;    // fetch value at pointer
```

16-bit dereference in assembly

```
   ldy  pt   ; Y points to data
   ldd  0,y  ; 16-bit fetch at pointer
   std  out

;read 16-bit contents pointed to by Y
```



6.1.1. Indexed addressing mode

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

5-bit (-16 to +15),
9-bit (-256 to +127), or
16-bit

machine	opcode	operand	comment
`$6A5C`	`staa`	`-4,Y`	`[Y-4] = RegA`

Let **n,R** be the indexed address
fixed offset **n** and
index register **R** is the index register, then
EAR will be **R+n**.

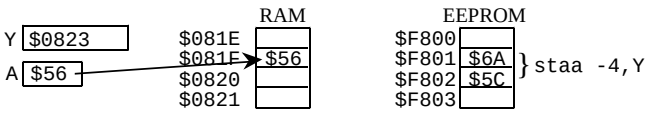


Figure 6.3. Example of the 9S12 indexed addressing mode.

16-bit data structures with indexed addressing is different in assembly versus in C.

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

The equivalent ROM-based definition is C would be

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

Want to fetch the 7 from **`Prime[4]`** In assembly,

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

or if we could have fetched it directly as

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

Either way, manipulating addresses in assembly always involves the physical byte-address regardless of the precision of the data.

Want to increment the pointer to the next element.

In C, we define the pointer as

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

and initialize it as

```
  Pt =  Prime;
```

To increment the pointer to the next element

in C, use the expression **`Pt++`**.

In assembly, we can define the pointer in RAM as

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

and initialize it as

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

However, to increment the pointer to the next element we have to add 2 to the pointer. E.g.,

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

6.1.2. Auto Pre/Post Decrement/Increment Indexed addressing mode

Optimized addressing modes to make it run fast

Not on Exam1

Not really needed for Lab

Regular access of an array

```
    staa 0,Y ;Store RegA at 2345,
    iny      ;Reg Y=2346
```

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

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

Regular access of an array, Y points to 16-bit element at 2344

```
    iny      ;Reg Y=2345
    iny      ;Reg Y=2346
    std  0,Y ;Store RegD at 2346,2347
```

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

```
    std 2,+Y ;Reg Y=2346, then store at 2346
```

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

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

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

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

6.1.3. Accumulator Offset Indexed addressing mode

Two registers combined to make effective address

One register points to array, other register has array index

Not on Exam1

Not really needed for Lab

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)
```

6.1.4. Indexed Indirect addressing mode

Optimized addressing modes for complex data structures

Not on Exam1

Not needed for Lab

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

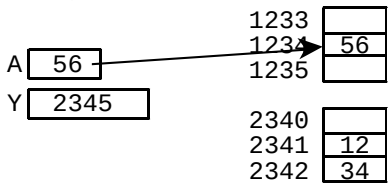


Figure 6.6. Example of the 9S12 indexed-indirect addressing mode.

6.1.5. Accumulator D Offset Indexed Indirect addressing mode

Optimized addressing modes for complex data structures

Not on Exam1

Not needed for Lab

```
    ldd  #4
    ldy  #2341
    stx  [D,Y]
;Store copy of value in Reg X at 1234 (D & Y unchanged)
```

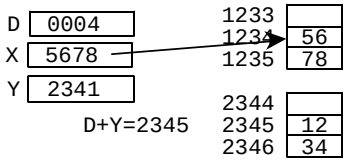


Figure 6.7. Example of the 9S12 accumulator-offset indexed-indirect addressing mode.

6.1.6. Post-byte machine coded for indexed addressing

For more information see Tables 6.1 and 6.2 in the book

Show xb- table

1) Open CPU12rg.pdf

2) Click on Indexed Addressing Mode Postbyte Encoding (xb)

6.1.6. Load effective address instructions

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

In each of the following cases, the effective address, **EA**, is loaded into Register X.

```
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 -16 to +15
q is any signed 9-bit -256 to +255
v is any unsigned 3 bit 1 to 8
W is any signed 16-bit -32768 to +32767 or any unsigned 16-bit 0 to 65535

4.5. 16-bit timer

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	
$0044																	TCNT

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$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

Table 4.11. 9S12 timer ports.

				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	667 kHz
1	1	0	64	8 µs	125 kHz	2.67 µs	333 kHz
1	1	1	128	16 µs	62.5 kHz	5.33 µs	167 kHz

Table 4.12. Given an E clock frequency, the PR2 PR1 and PR0 bits define the TCNT rate.

```
; 9S12DP512 at 8 MHz
; Enable TCNT at 1us
Timer_Init
   movb #$80,TSCR1  ;enable
   movb #$03,TSCR2  ;divide by 8
   rts
```

6.11. Functional Debugging

6.11.1. 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;
```

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++;
  }
}
```

Similar to Program 6.37. Instrumentation dump.
Next, you add **jsr Save** statements at strategic places within the system.
Use the debugger to display the results after program is done

6.2. Arrays

Random access
Sequential access.
An **array**
equal precision and
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.

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

In the previous examples, the length of the array was known.
One simple mechanism saves the length of the array as the first element.

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

We could define these variable length arrays in assembly as

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

Another common mechanism to handle variable length is a termination code.

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 transmission block

Table 6.3. Typical termination codes

6.3. Strings

A **string** is a data structure with
equal size elements
that only allows sequential access.

Example 6.4. 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`**, shown in Program 6.6, 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.

```
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. A variable length string contains ASCII data.

The bottom line

Pointers are addresses
Indexed addressing mode used for pointers
Precision: 8 bit or 16-bit
Arrays and strings have equal precision elements
TCNT is a 16-bit free-running clock