

# NET3001

C Programming

# Advantages

- programmers can work at a higher level than assembly language
  - can ignore the low level details
  - work at a higher level of abstraction
- code is (mostly) portable
- language provides
  - type checking
  - control statements other than test-&-jump
  - subroutine parameter/temp/return management
- still has low overhead
  - can still access pointers

# Data Types

<code>void</code>	functions that return nothing
<code>char</code>	single byte of data
<code>int/short</code>	single word of data
<code>long</code>	two words, 32 bit integer
<code>unsigned char</code>	control the way math operates on larger
<code>unsigned int</code>	numbers
<code>boolean</code>	a byte
<code>char[]</code>	an array of bytes
<code>float</code>	32 bit single-precision floating point
<code>double</code>	64 bit
	we won't be using floating point
<code>pointers</code>	to be discussed

# Logical Operations

- any expression which is non-zero is considered *true*

`(i<10)`

`(i>0 && i<10)`

`(i)`                    same as `(i!=0)`

- bit-wise operations

– only for char, short, int & long

```
P3OUT = P3OUT | 0x40; // set bit 6
```

```
P3OUT = P3OUT & 0xFE; // clear bit 0
```

```
P3OUT &= 0xFE; // the same
```

```
i = (P3IN >> 2); // shift right
```

```
if (!(P3IN & 0x80)) // check bit 7
```

# Operators

- math

`+ - * /`

`% (modulus)`

`++ -- (inc dec)`

- bit-wise operations

– returns char, short, int & long

`& | ^ << >>` (and or xor shift-l shift-r)

`~` (invert each bit)

- logical operations

– return true or false/0

`&& || == !=` (and or equal not-equal)

`> < >= <=` (math tests)

`!` (logical inversion)

# Flow Control

- C has a powerful set of control structures

```
if (condition) {  
    doTrueActions();  
} else {  
    doFalseActions();  
}
```

```
while(condition) {  
    doActions();  
}
```

```
do {  
    Actions();  
} while (condition);
```

// does at least once!

# Flow Control

- C has a powerful set of control structures

```
for (i=0; i<100; i++) {  
    doLoopActions ();  
}
```

// is

```
for (init-expression; test-expression; incr-expr)  
{  
    loopCode ();  
}
```

- init-expression is always executed
- test-expression is always executed
  - if the return value is true, then we loop
- the incr-expression is executed *after* the loop code has been run

# Flow Control

- multiple choice

```
switch (nChoice) {  
    case 0:  
        doFor0 ();  
        break;  
    case 2:  
        doFor2 ();  
        break;  
    case 7:  
        doFor7 ();  
        break;  
}
```

# Flow Control

- keywords which affect flow

## break

```
// escape out of the inner most for,while,do or switch
```

```
for (i=0; i<10; i++) {  
    r = sendPacket(i);  
    if (r==ERROR)  
        break;  
}
```

## continue

```
// finish the current for, while or do loop,  
//but keep executing
```

```
for (i=0; i<100; i++) {  
    if (packetType[i] == PACKET_SKIP)  
        continue;  
    sendPacket(i);  
}
```

# Pointers

- a pointer is another *type* of variable
- can address any *type* of variable
  - byte, int, array, function....anything
- a pointer is always 2 bytes long

```
char b;           // is 1 byte
&b;              // the address of 'b'
char* pb;        // is two bytes
pb = &b;         // write 2 bytes to pb
b = 4;           // write to the byte 'b'
*pb = 12;        // write to the SAME byte
```

```
int c;           // is 2 bytes
int* pc = &c;    // is 2 bytes, just a coincidence
```

```
char string1[14]; // is 14 bytes
char * string2;   // is 2 bytes
```

# Pointer Syntax

- '&' means “take the address of...”
  - will always be 2 bytes
- '\*' means “use this as an address, and go *through it* to access the data”
  - dereferencing
  - may be different sizes or properties, depending on the *type* of the pointer

```
unsigned char* puc;
```

```
a = 14 + *puc ; // *puc is an unsigned char
```

```
if (*puc < 150) // the compiler will do  
                // unsigned math on this
```

# Pointers

- on this CPU, pointers are 2 bytes
- pointers to writable data will be
  - around 0x200 for globals and statics
  - around 0x5E0 for variables on the stack
- pointers to const data will be
  - around 0x8000 (flash)

# Arrays and Pointers

- arrays are of size  $n * \text{type\_size}$
- pointers are always 2 bytes
- you can dereference them in two ways!

```
char aData[12] = {4,5,7,3,4, 6,5,4,3,2, 0,1};  
char* pData;          // is 2 bytes
```

```
// dereferencing  
aData[2];             // is one byte long ==7  
*(aData + 2);        // is the same ==7
```

```
// at this point, pData has 0 in it  
// assigning a pointer  
pData = aData;       // 2 bytes are written  
*(pData + 5);        // is a byte == 6  
pData[5];            // is the same == 6
```

# Pointers and Arrays

- arrays names go into the symbol table **and** take up  $n$  bytes in memory
- pointers go into the symbol table **and** take up 2 bytes in memory

```
char aData[12] = {4,5,7,3,4, 6,5,4,3,2, 0,1};  
char* pData;      // is 2 bytes
```

```
pData = aData;    // assign the 2 byte address  
pData++;         // modify the 2 bytes to point to next
```

```
aData++;         // !! not allowed!! illegal  
                // aData is the address of the 12 bytes  
                // you cannot change aData; only *aData
```

symbol	value	type	size	contents
aData	0x200	char[]	12	4,5,7,3,4.....
pData	0x20C	char*	2	0x200



# C to Assembler

- real example

```
char release = 1; // .byte 1 [in .data]
int i;          // .word 0 [in .data]

while (release != 0) // cmp #0, &release
    {}              // jnz $-4

P3DIR = 0x07;      // mov.b #7, &0x001A

while(1) {
    P3OUT = (i & 7); // mov &i, r15
                    // and #7, r15
                    // mov.b r15, &0x0019

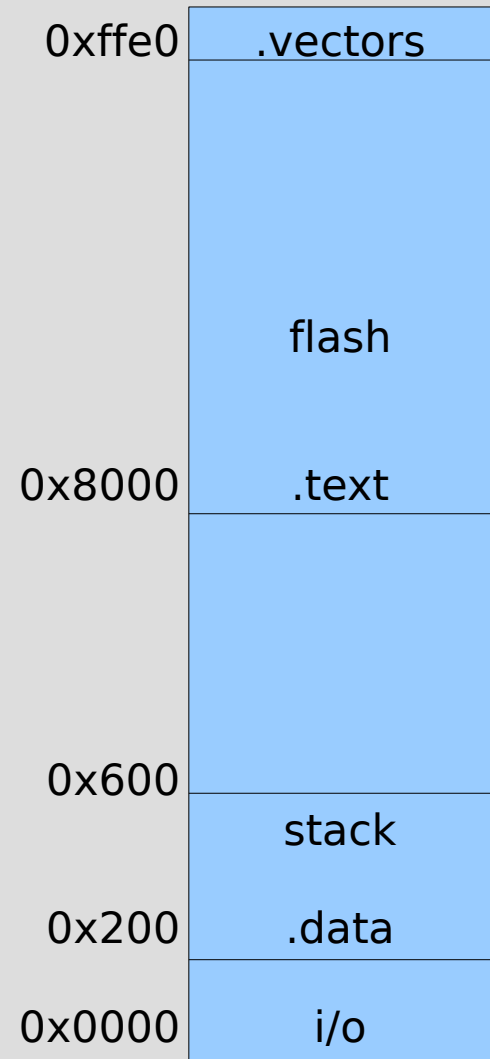
    i++;           // inc &i
    delay();       // call #delay
}                 // jmp $-20
```

# Memory Layout

```
char b = 4;           // goes into .data
char s;              // goes into .bss
const char f = 12;   // goes in .text

int main() {         // goes into .text
    int a;           // goes on the stack

    doStuff();
}
```



# Static

- the static keyword has several meanings
  - static on a function
    - visible only to *this* file
  - static on a variable outside a function
    - visible only to *this* file
  - static on a variable inside a function
    - variable exists even when the function exits
    - it lives in .bss or .data, has permanent address

```
int countDogs(int newDogs) {  
    static int nDogs;  
    nDogs = nDogs + newDogs;  
    return nDogs;  
}
```

# Data Areas

```
int a;           // Global in .bss
int b = 1;       // Initialized global .data
static int c;    // Static External in .bss
                // (only for this file)
static int d = 2; // Init'd static in .data

void anyFunction () { // Including main()
    int e;           // Automatic or Local in stack
    int f = 3;       // Automatic or Local in stack
    static int g;    // Persistent Local in .bss but only
                    // visible in function
    static int h=5;  // Init'd persistent Local in .data
                    // but only visible in function
}

const char[] str = "this is a string";
                // stored in the .text section
```

# C Defines

- #define lets you make your code easier to read

```
P2OUT |= 0x4;    // beep
delay_ms(2);
P2OUT &= ~0x4;
//-----or-----
#define BEEPER_ON_BIT 0x04
#define BEEPER_OFF_BIT ~0x04 // 0xFB
#define MAIN_OUT P2OUT

MAIN_OUT |= BEEPER_ON_BIT;
delay_ms(2);
MAIN_OUT &= BEEPER_OFF_BIT;
//-----or----- I like this one best
#define BEEPER_ON P2OUT |= 0x04
#define BEEPER_OFF P2OUT &= ~0x04

BEEPER_ON;
delay_ms(2);
BEEPER_OFF;
```

# Quiz

- what's wrong with this?

```
unsigned int w;  
// find the absolute value  
if (w<0)  
    w = -w;
```

- and this?

```
int doubleMe(int r) {  
    r = 2*r;  
}  
int x = 14;  
doubleMe(x);
```

# Symbol Table

```
int aVariable;
void one_sec_delay(void) {
    int i;
    for (i=0; i<30000; i++)
        ;
}
int main(void) {
    int r;
    P3DIR = 7;
    for (r=0; r<8; r++) {
        P3OUT = r;
        delay();
    }
}
```

# Symbol Table

Name	Value	Type	Size	Section	Content
aVariable	0x200	int	2	.bss	0
delay	0x8040	function void(void)	0	.text	
delay.i	stack 0(r4)	int	2	stack	
main	0x8052	function int(void)	0	.text	
main.r	stack 0(r4)	int	2	stack	

notice the focus on the **address** of the variables  
this is reflected in the fact that most assembly language instructions deal  
with the **address** of the variable

in actual fact, the variables delay.i and main.r may be stored in on  
the stack frame or they may be stored in registers; the compiler decides

# Symbol Table

```
int r;
/* set the LED pins to outputs */
P3DIR = 0x07;
8070:      f2 40 07 00      mov.b    #7,      &0x001a ;#0x0007
8074:      1a 00

/* 8 times, put a different pattern on the LED's and the
pause */
for(r=0; r<8; r++) {
8076:      84 43 00 00      mov     #0,      0(r4)      ;r3 As==00
807a:      b4 92 00 00      cmp     #8,      0(r4)      ;r2 As==11
807e:      01 38              jl      $+4      ;abs 0x8082
8080:      07 3c              jmp     $+16     ;abs 0x8090
      P3OUT = r;
8082:      e2 44 19 00      mov.b   @r4,     &0x0019 ;
      one_sec_delay();
8086:      b0 12 40 80      call   #-32704 ;#0x8040
808a:      94 53 00 00      inc    0(r4)    ;
808e:      f5 3f              jmp     $-20    ;abs 0x807a
}
```

# Startup Code

- disable the watchdog
- copy `.data` from flash to ram
  - ram contents are unknown/random at power-up time
- clear `.bss`
  - immediately after the `.data` section
- C++ constructors
- jump to `main()`
  
- `main()` must establish the stack pointer

# Common Idioms in C

- to do something  $n$  times

```
for (i=0; i<n; i++) {...}
```

- to wait for a bit to change

```
while ((P2IN & 1) == 0) // keyboard
    {}
```

```
while (TAR < 10000) // timer
    {}
```

- to count and loop back

```
while(1) // forever-loop
    for(i=0; i<100; i++)
        {}
```

- or

```
while(1) {
    i++;
    if (i==100)
        i=0;
}
```

# Program Examples

- write the code for an electric toothbrush
- I/O
  - single push button (P1.0)
  - the reciprocating motor (P1.2)
  - a timer (on chip), counting at 0.5 sec (TAR)
- functions
  - push button to turn on
  - push button to turn off
  - after 2 minutes of being on, turn the motor off/on/off/on/off/on at 0.5 second intervals



# Program Examples

- write the code for a model railroad crossing
- I/O
  - single train sensor (P1.0)
  - the motor to raise the gate (P1.2)
  - motor to lower the gate (P1.3)
  - two red LED's (P1.4 & P1.5)
  - noise maker (P1.7)
  - a timer (on chip), counting at 0.1 sec (TAR)
- functions
  - when a train is present
    - lower the gate (motor on for 1 sec)
    - flash the two LED's alternately, at 0.5 seconds
    - operate the dinger, at 8 dings/sec
  - when the train leaves
    - stop LED's and dinger
    - raise the gate (motor on the other way for 1 sec)

