C Programming

Revisit Timer Exercise

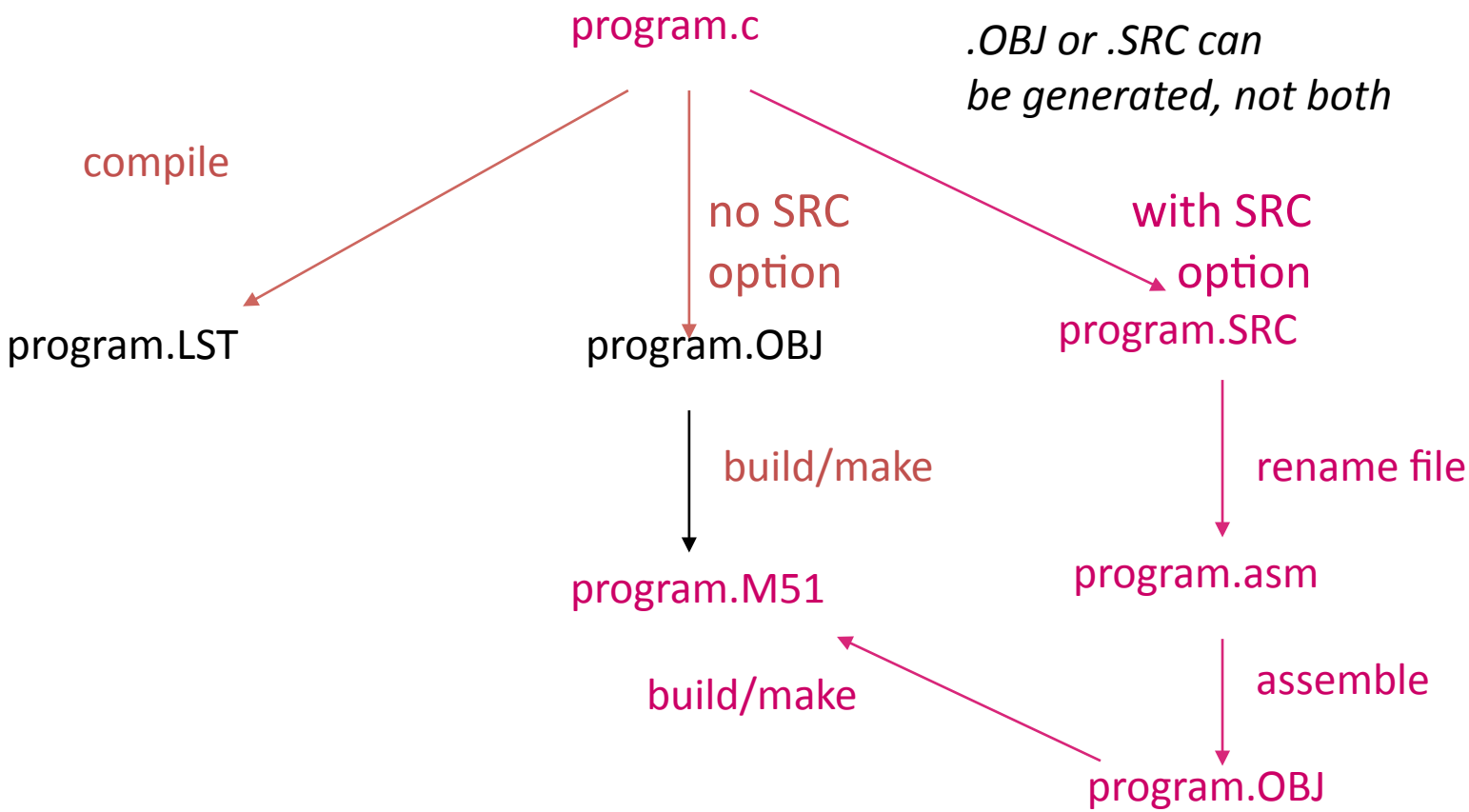
Blinking!

In-line Assembly

- When it is more efficient, or easier, can insert assembly code in C programs.

```
#pragma asm
put your assembly code here
#pragma endasm
```

Compilation Process (Keil)



Must use this path for C programs with in-line assembly
It is also necessary to add #pragma SRC to code

Example – Switch/LED Program

```
#include <c8051F020.h>
#pragma SRC                          // Need this to generate .SRC file
void PORT_Init (void);

char Get_SW(void) {
#pragma ASM
mov a, P3
anl a, #80h                  ; mask all but P3.7
mov R7, a                    ; function value (char) returned in R7
#pragma ENDASM
}

void Set_LED(void) {
#pragma ASM
setb P1.6
#pragma ENDASM
}

void Clr_LED(void) {
#pragma ASM
clr P1.6
#pragma ENDASM
}
void PORT_Init (void){   XBR2    = 0x40;        // Enable crossbar and enable P1.6 (LED) as push-pull output}
P1MDOUT |= 0x40;                              // enable P1.6 (LED) as push-pull output
}
void main(void) {
PORT_Init();
while (1)
if (Get_SW())     Set_LED();
else     Clr_LED();
                  }
```

Functions can be implemented in assembly language

Main function

Interfacing with C

- Example: Temperature Sensor program
  - Configures the external oscillator
  - Configures the ADC0 for temp. sensor
  - Configures Port1 so LED can be used
  - Configures Timer3 to synch the ADC0
  - Uses ADC0 ISR to take temperature samples and averages 256 of them and posts average to global variable
  - Main program compares average temp. to room temp. and lights LED if temp is warmer.
  - Temp_2.c

Revisit DAC0 Program

And “C” the difference!

Converting to Real Values

- C makes it easier to implement equations

Example: Temperature conversion

For analog to digital conversion – assuming left justified:

$$V = \frac{ADC0/16}{2^{12}} \times \frac{Vref}{Gain}$$

The temperature sensor:

$$Temp_C = \frac{V - 0.776}{0.00286}$$

Temperature Conversion

$$Temp_C = \frac{(\frac{ADC0/16}{2^{12}} \times \frac{Vref}{Gain}) - 0.776}{0.00286}$$

Let Vref = 2.4V, Gain = 2

$$Temp_C = \frac{ADC0 - 42380}{156}$$

C for the Equation

$$Temp_C = \frac{ADC0 - 42380}{156}$$

```
…
unsigned int result, temperature;

…
result = ADC0;                          //read temperature sensor
temperature = result - 42380;
temperature = temperature / 156;
```

* Must be careful about range of values expected and variable types

Make it REAL!

Temperature Conversion

Initialization

- When a C program is compiled, some code is created that runs BEFORE the main program.
- This code clears RAM to zero and initializes your variables. Here is a segment of this code:

```
        LJMP 0003h
0003:   MOV R0, #7FH
        CLR A
back:   MOV @R0, A
        DJNZ  R0, back
...
```

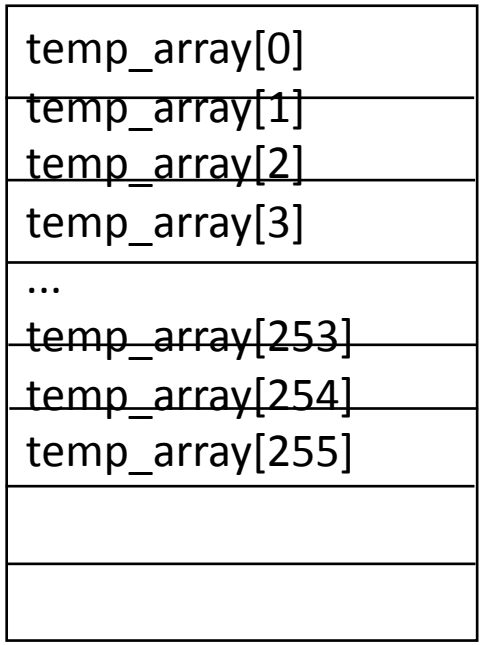
Arrays in C

- Useful for storing data

type arr_name[dimension]

char temp_array[256]

Array elements are stored in adjacent locations in memory.



Pointers in C

- Pointers are variables that hold memory addresses.
- Specified using * prefix.

```
int *pntr;        // defines a pointer, pntr
pntr = &var;   // assigns address of var to pntr
```

Pointers and Arrays

Note: the name of an array is a pointer to the first element:

*temp_array is the same as temp_array[0]

So the following are the same:

```
n = *temp_array;
n = temp_array[0];
```

and these are also the same:

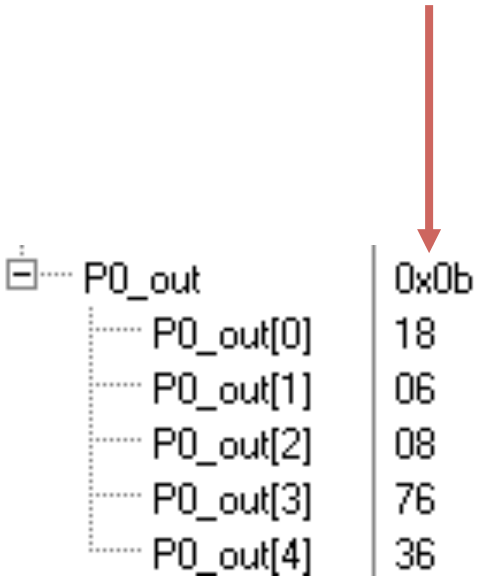
```
n = *(temp_array+5);
n = temp_array[5];
```

temp_array[0]
temp_array[1]
temp_array[2]
temp_array[3]
…

Arrays

- In watch window, address (pointer) of first element array is shown.
- Array is not initialized as you specify when you download or reset, but it will be when Main starts.

unsigned char P0_out[4] = {0x01,0x02,0x04,0x08};



Array Example

Compiler Optimization Levels

- Optimization level can be set by compiler control directive:
- Examples (default is #pragma (8, speed)
  - #pragma ot (7)
  - #pragma ot (9, size)
  - #pragma ot (size) – reduce memory used at the expense of speed.
  - #pragma ot (speed) – reduce execution time at the expense of memory.

Compiler Optimization Levels

Level	Optimizations added for that level
0	**Constant Folding:** The compiler performs calculations that reduce expressions to numeric constants, where possible.This includes calculations of run-time addresses. **Simple Access Optimizing:** The compiler optimizes access of internal data and bit addresses in the 8051 system. **Jump Optimizing:** The compiler always extends jumps to the final target. Jumps to jumps are deleted.
1	**Dead Code Elimination:** Unused code fragments and artifacts are eliminated. **Jump Negation:** Conditional jumps are closely examined to see if they can be streamlined or eliminated by the inversion of the test logic.
2	....
3	
4	
5	
6	
7	
8	
9	**Common Block Subroutines:** Detects recurring instruction sequences and converts them into subroutines. **C*x*51** evenrearranges code to obtain larger recurring sequences.

Example: 7-seg Decoder

```
// Program to convert 0-F into 7-segment equivalents.
#pragma debug code)
#pragma ot (9)
#include <c8051f020.h>
#define NUM_SAMPLES 16
unsigned char SEGS7[16] = {0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92,
0x82, 0xF8, 0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E};

xdata unsigned char samples[NUM_SAMPLES];

void main (void)
{
   char i;                        // loop counter
   WDTCN = 0xde;
   WDTCN = 0xad;
   for (i=0; i < NUM_SAMPLES; i++)
        {samples[i] = SEGS7[i];}
   while (1);
}
```

Effect of Optimization Level on Code Size

Level	Code Size
0	53
1	53
2	53
3	51
4	46
5	46
6	39
7	39
8	38
9	38

Level 0 Optimization

```
; FUNCTION main (BEGIN)
0000 75FFDE             MOV     WDTCN,#0DEH
0003 75FFAD             MOV     WDTCN,#0ADH
;---- Variable 'i' assigned to Register 'R7' ----
0006 750000      R      MOV     i,#00H
0009 C3                 CLR     C
000A E500        R      MOV     A,i
000C 6480               XRL     A,#080H
000E 9490               SUBB    A,#090H
0010 5020               JNC     ?C0004
0012 AF00        R      MOV     R7,i
0014 7400        R      MOV     A,#LOW SEGS7
0016 2F                 ADD     A,R7
0017 F8                 MOV     R0,A
0018 E6                 MOV     A,@R0
…
```

Level 9 Optimization

```
; FUNCTION main (BEGIN)
0000 75FFDE             MOV     WDTCN,#0DEH
0003 75FFAD             MOV     WDTCN,#0ADH
;---- Variable 'i' assigned to Register 'R7' ----
0006 E4                 CLR     A
0007 FF                 MOV     R7,A
0008 7400         R     MOV     A,#LOW SEGS7
000A 2F                 ADD     A,R7
000B F8                 MOV     R0,A
000C E6                 MOV     A,@R0
…
```

Memory Models

- Small - places all function variables and local data segments in the internal data memory (RAM) of the 8051 system. This allows very efficient access to data objects (direct and register modes). The address space of the **SMALL** memory model, however, is limited.
- Large - all variables and local data segments of functions and procedures reside (as defined) in the external data memory of the 8051 system. Up to 64 KBytes of external data memory may be accessed. This,however, requires the long and therefore inefficient form of data access through the data pointer (**DPTR**).
- Selected by compiler directives
- Examples:
  - #pragma small
  - #pragma large

Example: LARGE

```
0006 E4                 CLR     A
0007 FF                 MOV     R7,A
0008 EF                 MOV     A,R7
0009 FD                 MOV     R5,A
000A 33                 RLC     A       ;multiply by 2
000B 95E0               SUBB    A,ACC
000D FC                 MOV     R4,A
000E 7400         R     MOV     A,#LOW SEGS7
0010 2D                 ADD     A,R5
0011 F582               MOV     DPL,A
0013 7400         R     MOV     A,#HIGH SEGS7
0015 3C                 ADDC    A,R4
0016 F583               MOV     DPH,A
0018 E0                 MOVX    A,@DPTR
….
```

Registers R4, R5 keep track of 16-bit data address (external RAM)

Example: SMALL

```
0006 E4                 CLR     A
0007 FF                 MOV     R7,A
0008 7400          R    MOV     A,#LOW SEGS7
000A 2F                 ADD     A,R7
000B F8                 MOV     R0,A
000C E6                 MOV     A,@R0
….
```

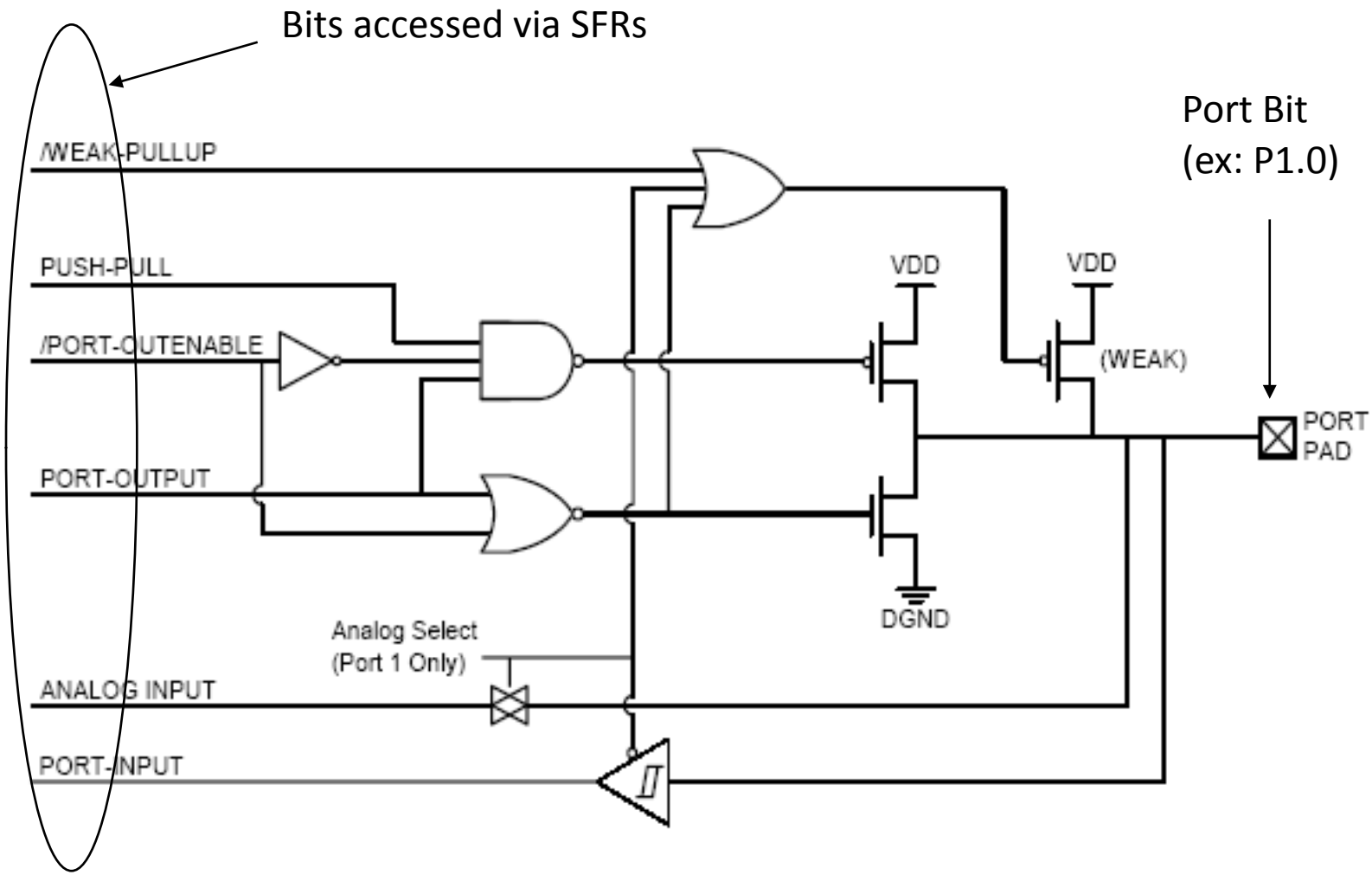
Data address = #LOW SEGS7 + R7 (8-bit address, RAM)

Initialization

- When a C program is compiled, some code is created that runs BEFORE the main program.
- This code clears RAM to zero and initializes your variables. Here is a segment of this code:

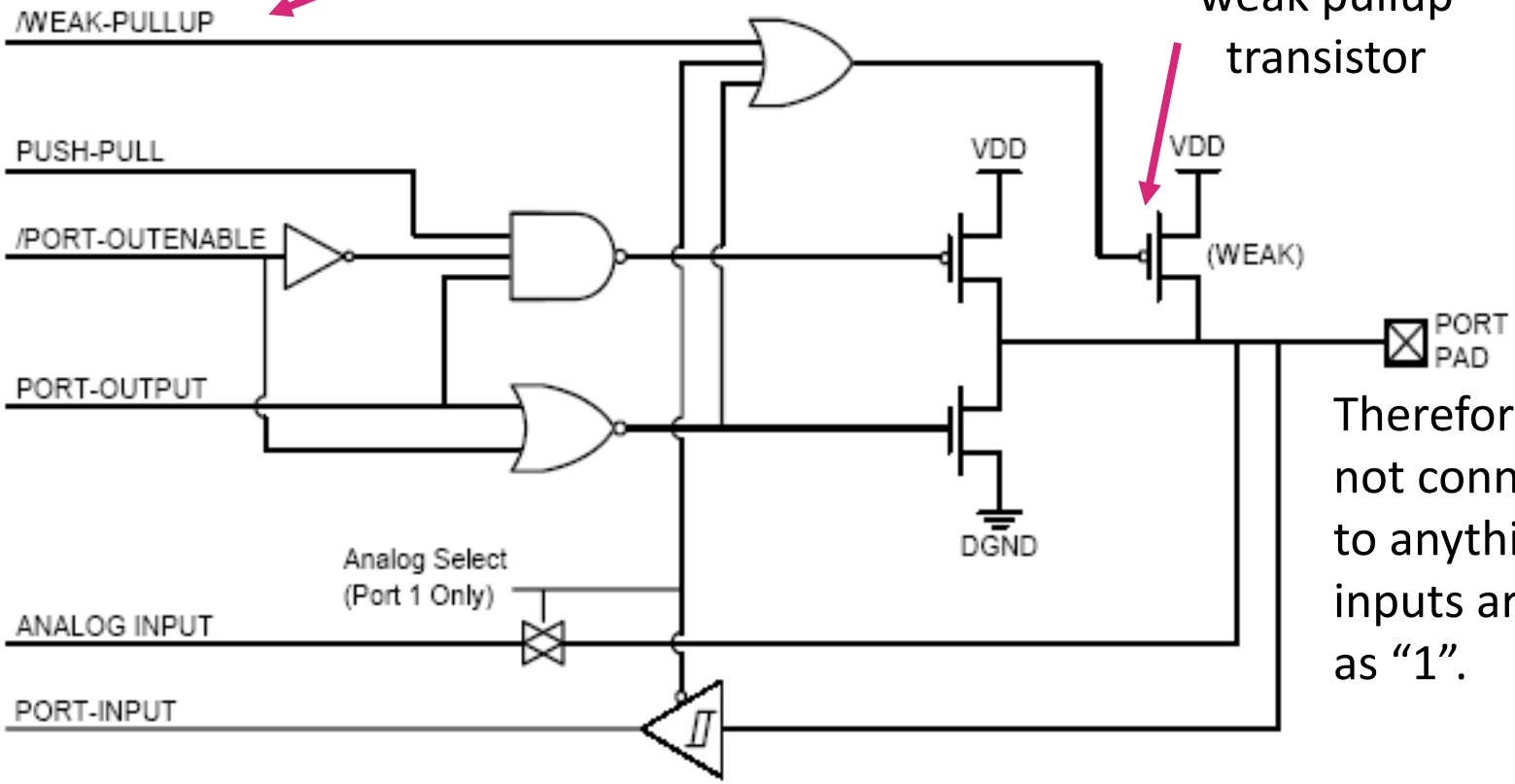
```
        LJMP 0003h
0003:   MOV R0, #7FH
        CLR A
back:   MOV @R0, A
        DJNZ  R0, back
...
```

I/O Circuitry - Exercise



Can be disabled.

By default, inputs are “pulled up” by weak pullup transistor

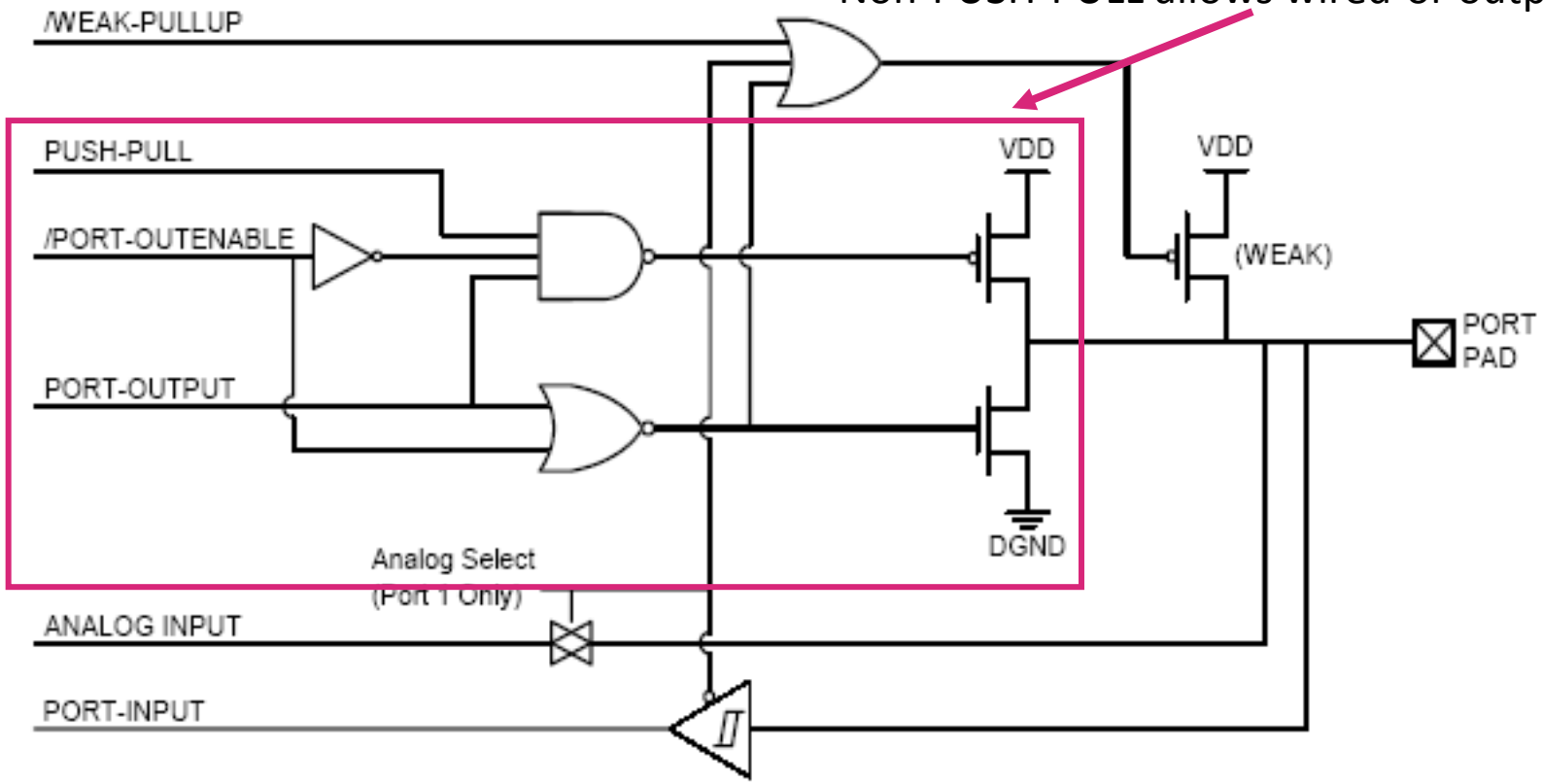


Therefore, if not connected to anything, inputs are read as “1”.

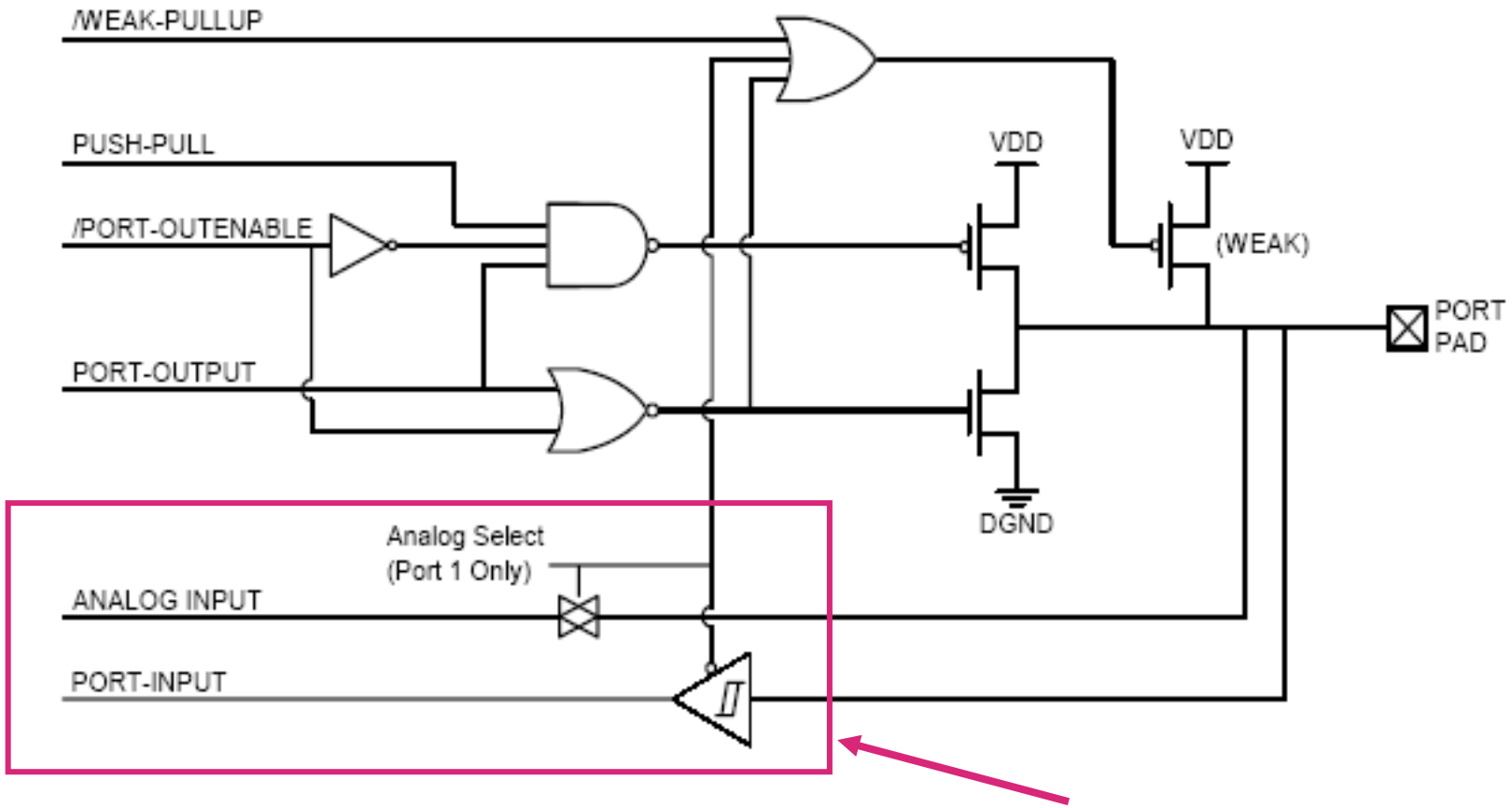
Port I/O - Output

Output circuit:

- Only enabled if /PORT-OUTENABLE = 0
- PUSH-PULL = 1 enables P transistor
- Non-PUSH-PULL allows wired-or outputs



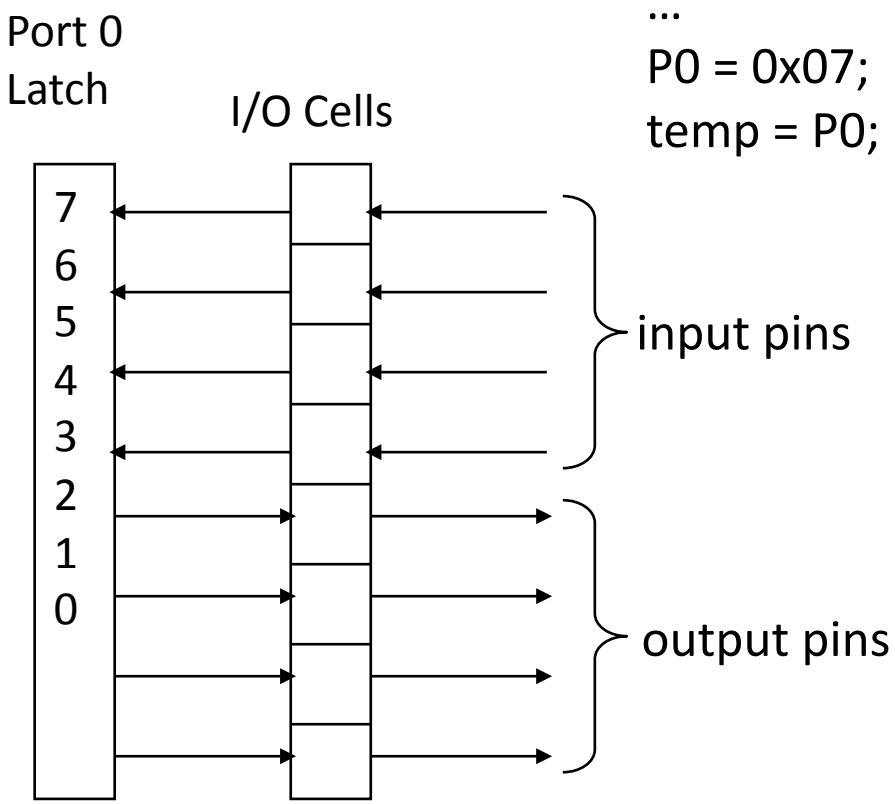
Port I/O - Input



Port 1 can be configured for either digital or analog inputs using a pass transistor and buffer

Port I/O Example

```
XBR2 = 0x40;              // Enable XBAR2
P0MDOUT = 0x0F;    // Outputs on P0 (0-3)
…
P0 = 0x07;      // Set pins 2,1,0 and clear pin 3
temp = P0;      // Read Port0
```



Keypad Interface

C for Large Projects

- Use functions to make programs modular
- Break project into separate files if the programs get too large
- Use header (#include) files to hold definitions used by several programs
- Keep main program short and easy to follow
- Consider multi-tasking or multi-threaded implementations

Functions

- The basis for modular structured programming in C.

```
return-type   function-name(argument declarations)
{
        declarations and statements
}
```

Example – no return value or arguments

```
void SYSCLK_Init (void) {
   // Delay counter
   int i;
            // Start external oscillator with 22.1184MHz crystal
            OSCXCN = 0x67;
            // Wait for XTLVLD blanking interval (>1ms)
            for (i = 0; i < 256; i++) ;
            // Wait for crystal osc. to settle
            while (!(OSCXCN & 0x80)) ;
            // Select external oscillator as SYSCLK
            OSCICN = 0x88;
}
```

Example – with arguments

```
void Timer3_Init (int counts) {
    // Stop timer, clear TF3, use SYSCLK as timebase
    TMR3CN = 0x02;
    // Init reload value
    TMR3RL = -counts;
    // Set to reload immediately
    TMR3 = 0xffff;
    // Disable interrupts
    EIE2 &= ~0x01;
    // Start timer
    TMR3CN |= 0x04;
}
```

Example – with return value

```
char ascii_conv (char num) {
    return num + 30;
    }
```

Header Files

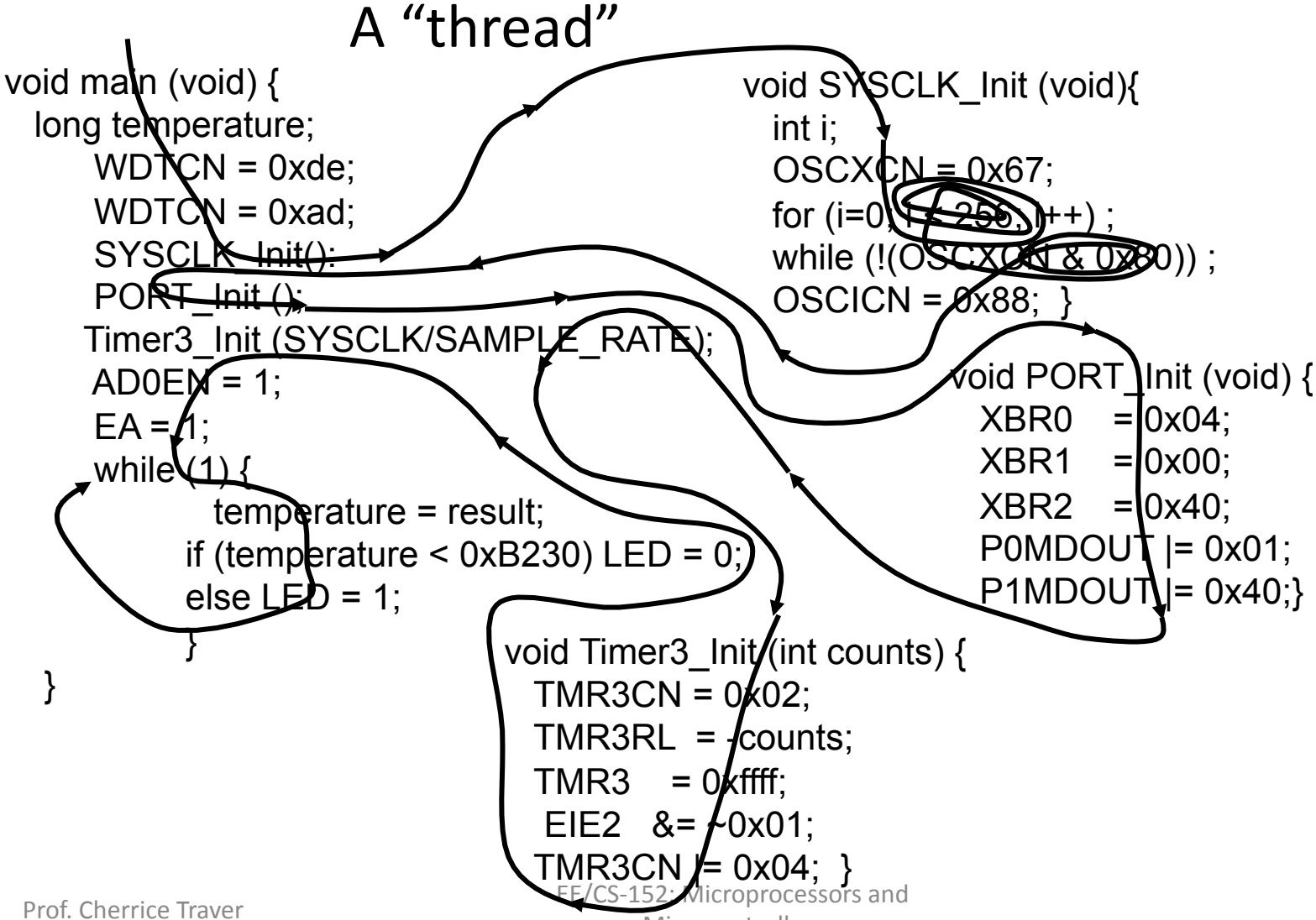
- Use to define global constants and variables

```
// 16-bit SFR Definitions for 'F02x
        sfr16 TMR3RL = 0x92;                // Timer3 reload value
        sfr16 TMR3 = 0x94;                  // Timer3 counter
        sfr16 ADC0 = 0xbe;                  // ADC0 data
        sfr16 DAC0 = 0xd2;                  // DAC data
        sfr16 DAC1 = 0xd5;
// Global CONSTANTS
        #define SYSCLK 22118400             // SYSCLK frequency in Hz
        sbit LED = P1^6;                    // LED='1' means ON
        sbit SW1 = P3^7;                    // SW1='0' means switch pressed
        #define MAX_DAC ((1<<12)-1)         // Maximum value of the DAC register 12 bits
        #define MAX_INTEGRAL (1L<<24)       // Maximum value of the integral
// Function PROTOTYPES
        void SYSCLK_Init (void);
        void PORT_Init (void);
        void ADC0_Init (void);
        void DAC_Init (void);
        void Timer3_Init (int counts);
        void ADC0_ISR (void);
```

Multitasking and Multithreading

- Multitasking: Perception of multiple tasks being executed simultaneously.
  - Usually a feature of an operating system and tasks are separate applications.
  - Embedded systems are usually dedicated to one application.
- Multithreading: Perception of multiple tasks within a single application being executed.
  - Example: Cygnal IDE color codes while echoing characters you type.

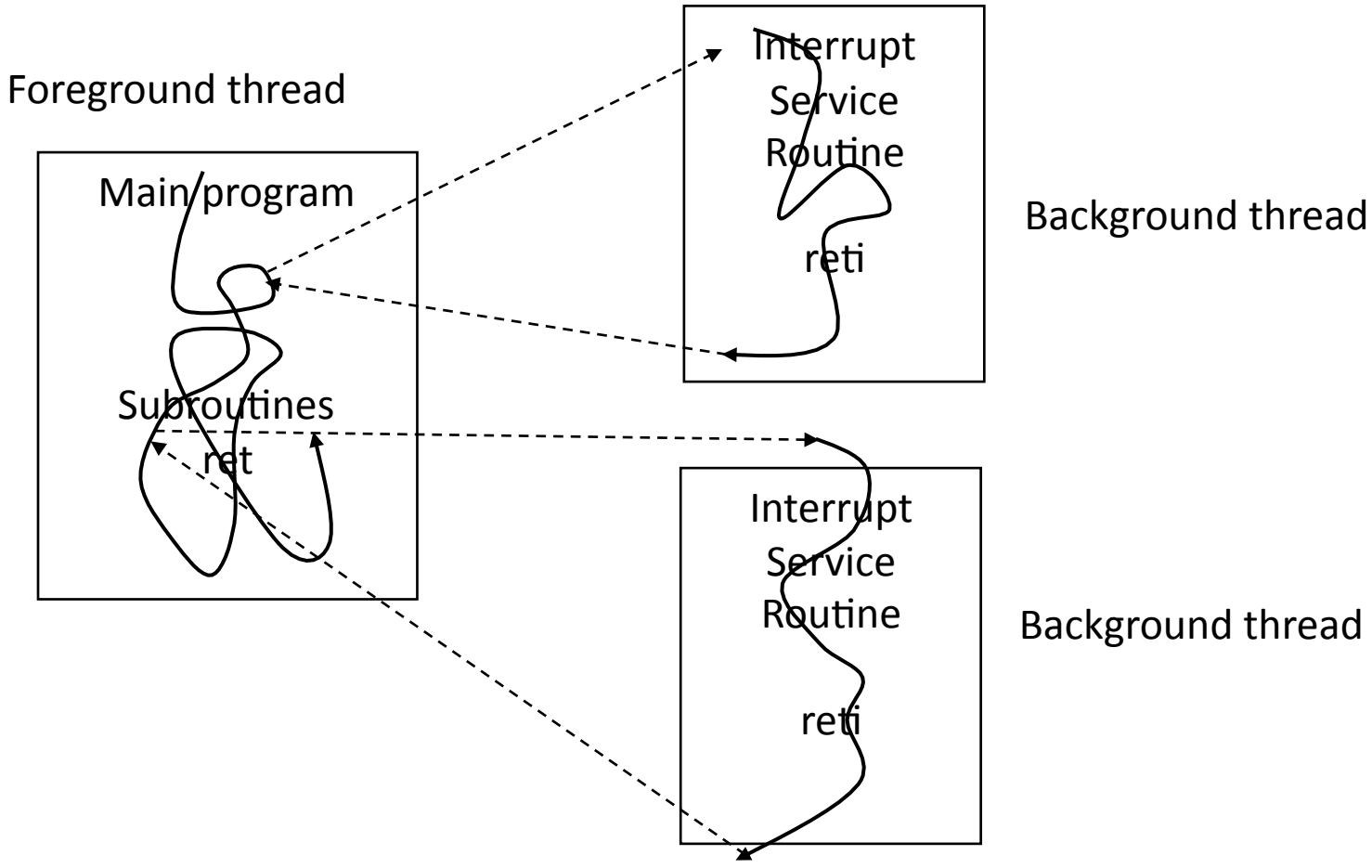
Multitasking and Multithreading



Multi-tasking/threading Implementations

- Cooperative multi-tasking – each application runs for a short time and then yields control to the next application.
- Timer-based multi-tasking – on each timer interrupt, tasks are switched.
- When switching between tasks, state of processor (internal registers, flags, etc) must be saved and previous state from last task restored. This is the "overhead" of multitasking. Also called "context switching".

Multithreading with Interrupts



Real-Time Operating Systems (RTOS)

- Usually a timer-based task switching system that can guarantee a certain response time.
- Low level functions implement task switching.
- High level functions create and terminate threads or tasks.
- Each task might have its own software stack for storing processor state.