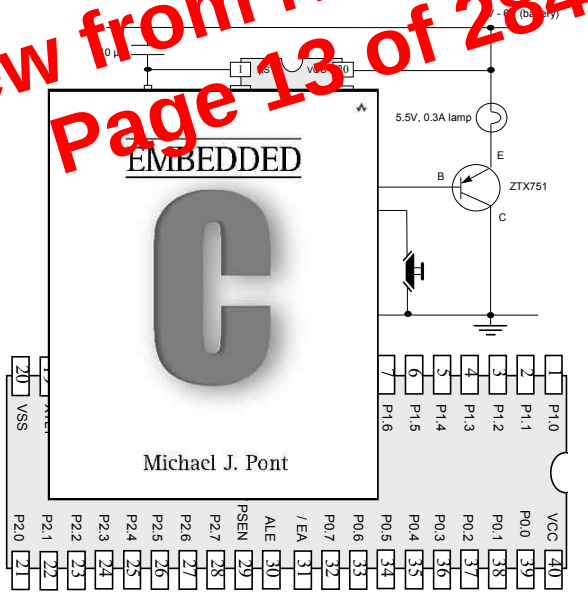
Seminar 1:
"Hello, Embedded World"



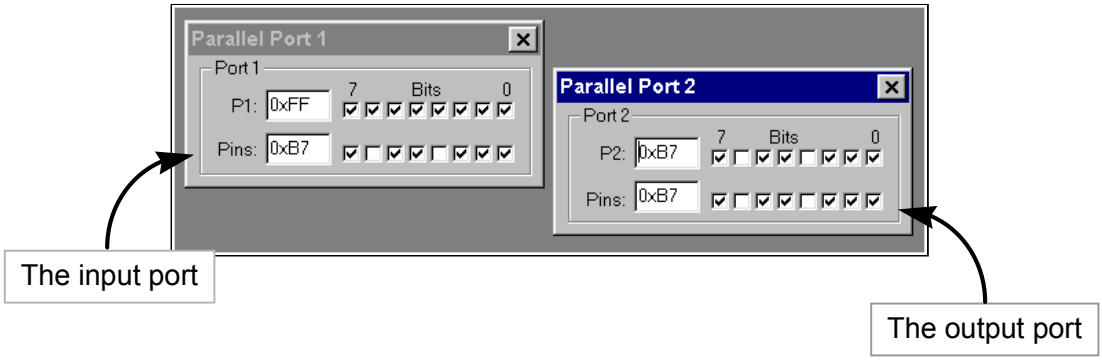
 Contains material from:
Pont, M.J. (2002) "Embedded C", Addison-Wesley.

By the end of the course …

By the end of the course, you will be able to:

1. Design software for single-processor embedded applications based on small, industry standard, microcontrollers;
2. Implement the above designs using a modern, high-level programming language (‘C’), and
3. Begin to understand issues of reliability and safety and how software design and programming decisions may have a positive or negative impact in this area.

Example: Reading and writing bytes




```
void main (void)
   {
   unsigned char Port1_value;

   /* Must set up P1 for reading */
   P1 = 0xFF;

   while(1)
     {
     /* Read the value of P1 */
     Port1_value = P1;

     /* Copy the value to P2 */
     P2 = Port1_value;
     }
   }
```

Strengths and weaknesses of software-only delays

☺ SOFTWARE DELAY can be used to produce very short delays.

☺ SOFTWARE DELAY requires no hardware timers.

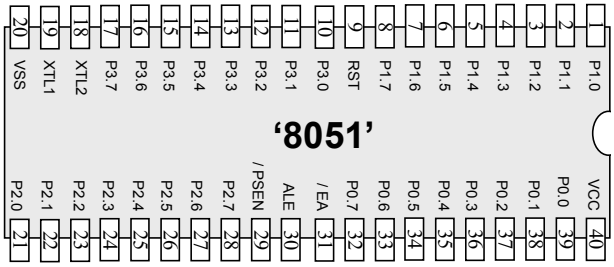
☺ SOFTWARE DELAY will work on any microcontroller.

BUT:

☹ It is very difficult to produce precisely timed delays.

☹ The loops must be re-tuned if you decide to use a different processor, change the clock frequency, or even change the compiler optimisation settings.

Review: The 8051 microcontroller

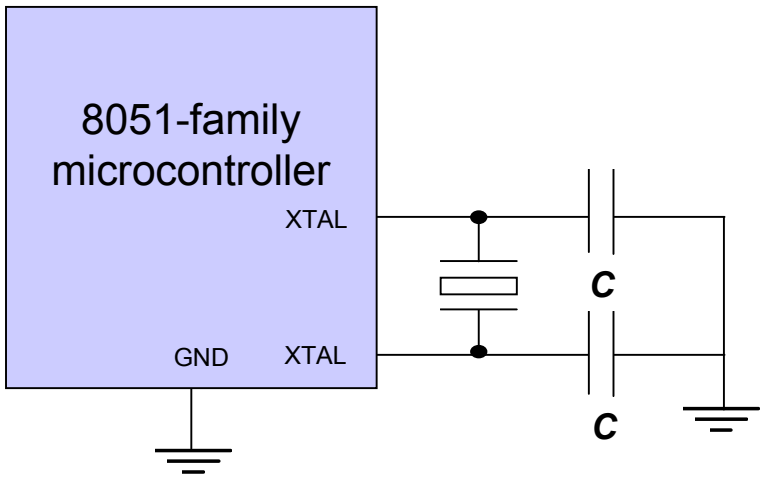


Typical features of a modern 8051:

- Thirty-two input / output lines.
- Internal data (RAM) memory - 256 bytes.
- Up to 64 kbytes of ROM memory (usually flash)
- Three 16-bit timers / counters
- Nine interrupts (two external) with two priority levels.
- Low-power Idle and Power-down modes.

The different members of this family are suitable for everything from automotive and aerospace systems to TV "remotes".

How to connect a crystal to a microcontroller



In the absence of specific information, a capacitor value of 30 pF will perform well in most circumstances.

Overall strengths and weaknesses

- ☺ Crystal oscillators are stable. Typically ±20-100 ppm = ±50 mins per year (up to ~1 minute / week).
- ☺ The great majority of 8051-based designs use a variant of the simple crystal-based oscillator circuit presented here: developers are therefore familiar with crystal-based designs.
- ☺ Quartz crystals are available at reasonable cost for most common frequencies. The only additional components required are usually two small capacitors. Overall, crystal oscillators are more expensive than ceramic resonators.

BUT:

- ☹ Crystal oscillators are susceptible to vibration.
- ☹ The stability falls with age.

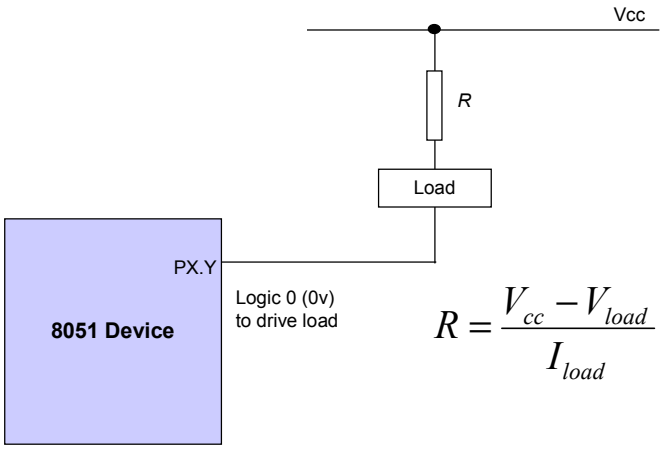
Driving DC Loads

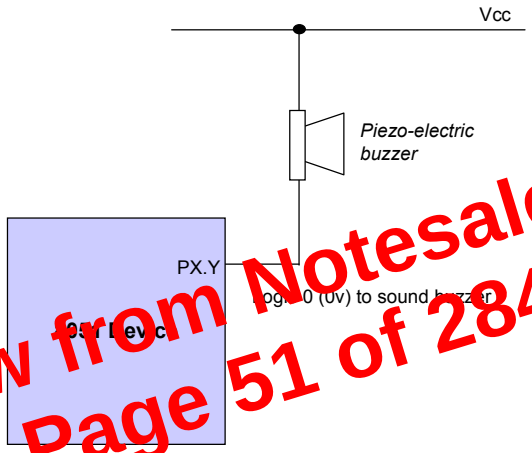
- The port pins on a typical 8051 microcontroller can be set at values of either 0V or 5V (or, in a 3V system, 0V and 3V) under software control.

- Each pin can typically sink (or source) a current of around 10 mA.

- The total current we can source or sink per microcontroller (all 32 pins, where available) is typically 70 mA or less.

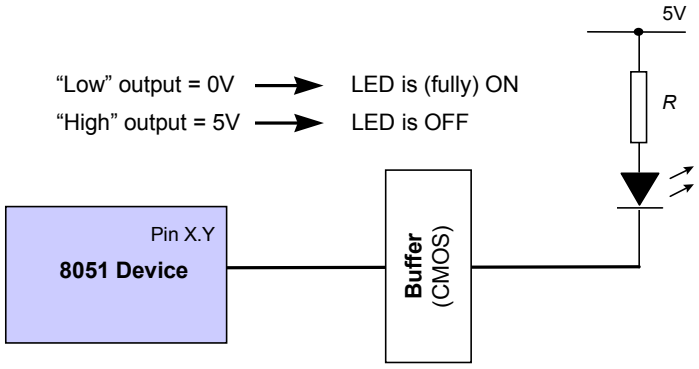
Driving a low-power load without using a buffer



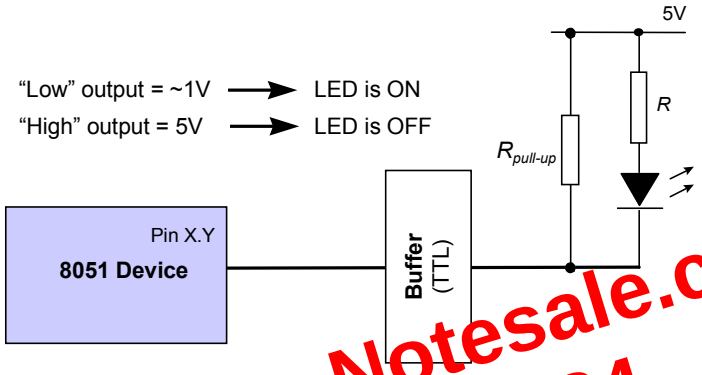


See "PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS", p.115
(NAKED LOAD)

Using an IC Buffer



Using a CMOS buffer.

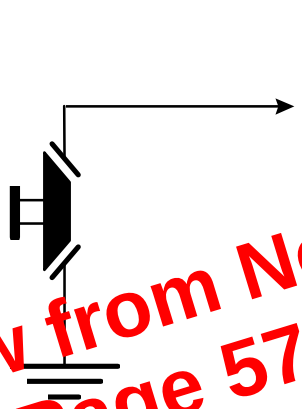


Using a TTL buffer.

It makes sense to use CMOS logic in your buffer designs wherever possible. You should also make it clear in the design documentation that CMOS logic is to be used.

See "PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS", p.118 (IC BUFFER)

Seminar 3:
Reading Switches



Example: Reading and writing bits (simple version)

```
/*-------------------------------------------------------------*-

   Bits1.C (v1.00)

-*-------------------------------------------------------------*/

#include <Reg52.H>

sbit Switch_pin = P1^0;
sbit LED_pin = P1^1;

/* ............................................................ */

void main (void)
   {
   bit x;

   /* Set switch pin for reading */
   Switch_pin = 1;

   while(1)
     {
     x = Switch_pin;    /* Read Pin 1.0 */
     LED_pin = x;       /* Write to Pin 1.1 */
     }
   }

/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------
-*-------------------------------------------------------------*/
```

Example: Reading and writing bits (generic version)

The six bitwise operators:

Operator	Description
&	Bitwise AND
|	Bitwise OR (inclusive OR)
^	Bitwise XOR (exclusive OR)
<<	Left shift
>>	Right shift
~	One's complement

A	B	A **AND** B	A **OR** B	A **XOR** B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

```
/* ------------------------------------------------------------ */

void Display_Byte(const unsigned char CH)
   {
   unsigned char i, c = CH;
   unsigned char Mask = 1 << 7;

   for (i = 1; i <= 8; i++)
      {
      putchar(c & Mask ? '1' : '0');
      c <<= 1;
      }

   putchar('\n');
   }
```

```
x                                   11111110
1s complement [~x]                  00000001
Bitwise AND [x & 0x0f]              00001110
Bitwise OR [x | 0x0f]               11111111
Bitwise XOR [x ^ 0x0f]              11110001
Left shift, 1 place [x <<= 1]       11111100
Right shift, 4 places [x >>= 4]     00001111


Display MS byte of unsigned int y   00001010
Display LS byte of unsigned int y   00001011
```

```
/* ------------------------------------------------------------ */

bit Read_Bit_P1(const unsigned char PIN)
   {
   unsigned char p = 0x01;  /* 00000001 */

   /* Left shift appropriate number of places */
   p <<= PIN;

   /* Write a 1 to the pin (to set up for reading) */
   Write_Bit_P1(PIN, 1);

   /* Read the pin (bitwise AND) and return */
   return (P1 & p);
   }

/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------
-*-------------------------------------------------------------*/
```

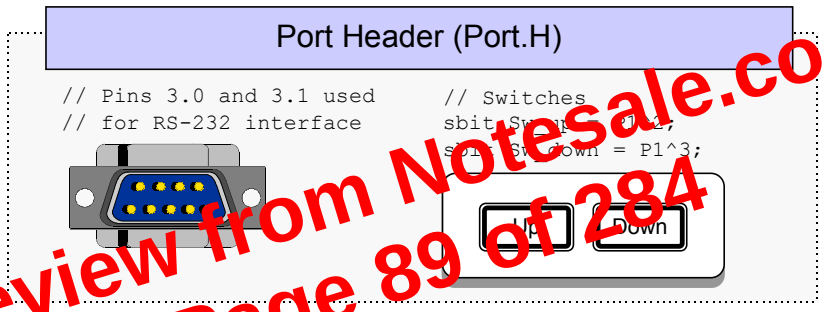
Conclusions

The switch interface code presented and discussed in this seminar has allowed us to do two things:

- To perform an activity while a switch is depressed;
- To respond to the fact that a user has pressed – and then released – a switch.

In both cases, we have illustrated how the switch may be 'debounced' in software.

Seminar 4:
Adding Structure to Your Code



```
/*-------------------------------------------------------------*/
void PC_LINK_IO_Write_Char_To_Buffer(...)
   {
   ...
   }

/*-------------------------------------------------------------*/
void PC_LINK_IO_Write_String_To_Buffer(...)
   {
   ...
   }

/*-------------------------------------------------------------*/
char PC_LINK_IO_Get_Char_From_Buffer(...)
   {
   ...
   }

/*-------------------------------------------------------------*/
void PC_LINK_IO_Send_Char(...)
   {
   ...
   }
```

Summary: Why use the Project Header?

Use of PROJECT HEADER can help to make your code more readable, not least because anyone using your projects knows where to find key information, such as the model of microcontroller and the oscillator frequency required to execute the software.

The use of a project header can help to make your code more easily portable, by placing some of the key microcontroller-dependent data in one place: if you change the processor or the oscillator used then - in many cases - you will need to make changes only to the Project Header.

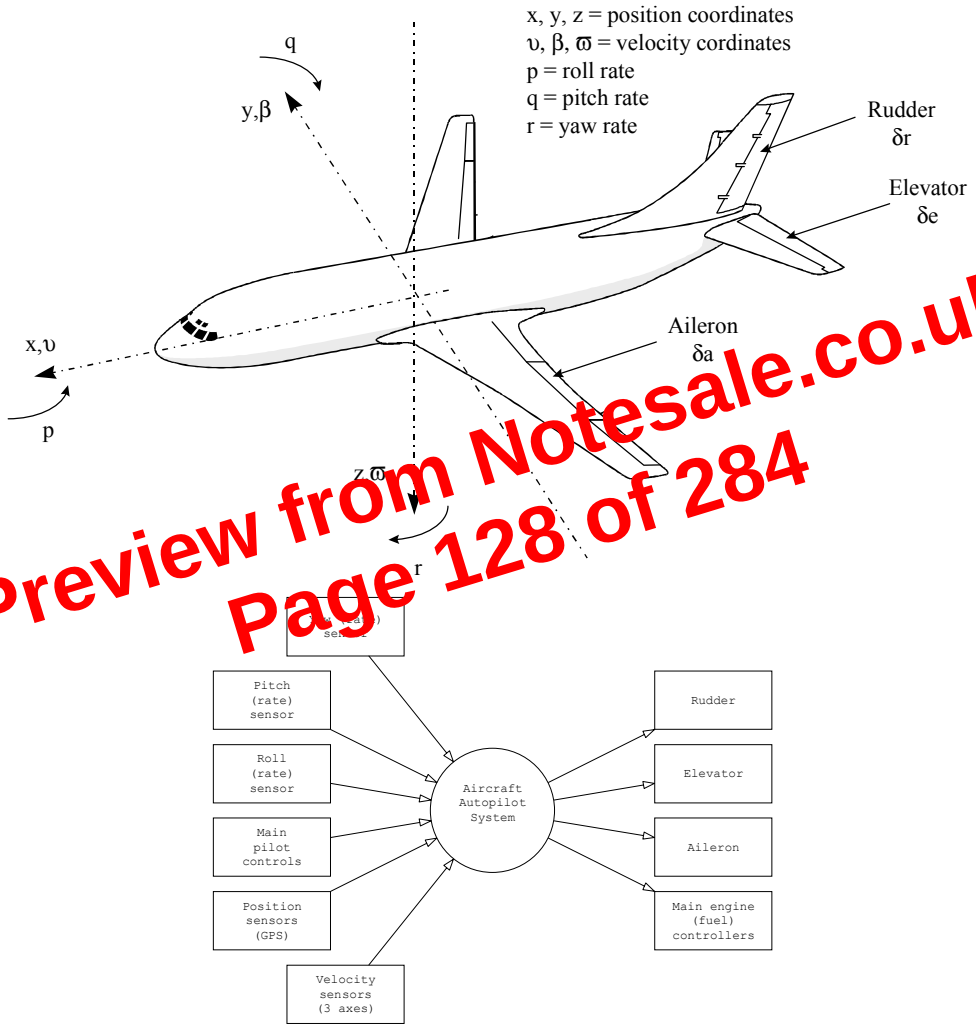
```
/*-------------------------------------------------------------*-

   Main.C (v1.00)

  ------------------------------------------------------------

   A "Hello Embedded World" test program for 8051.

   (Re-structured version - multiple source files)

-*-------------------------------------------------------------*/

#include "Main.H"
#include "Port.H"

#include "Delay_Loop.h"
#include "LED_Flash.h"

void main(void)
   {
   LED_FLASH_Init();

   while(1)
      {
      /* Change the LED state (OFF to ON, or vice versa) */
      LED_FLASH_Change_State();

      /* Delay for *approx* 1000 ms */
      DELAY_LOOP_Wait(1000);
      }
   }

/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------
-*-------------------------------------------------------------*/
```

```
/*-------------------------------------------------------------*-

   Delay_Loop.H (v1.00)

  ------------------------------------------------------------

   - See Delay_Loop.C for details.

-*-------------------------------------------------------------*/

#ifndef _DELAY_LOOP_H
#define _DELAY_LOOP_H

/* ------ Public function prototype --------------------------- */
void DELAY_LOOP_Wait(const tWord DELAY_MS);

#endif

/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------
-*-------------------------------------------------------------*/
```

Introduction

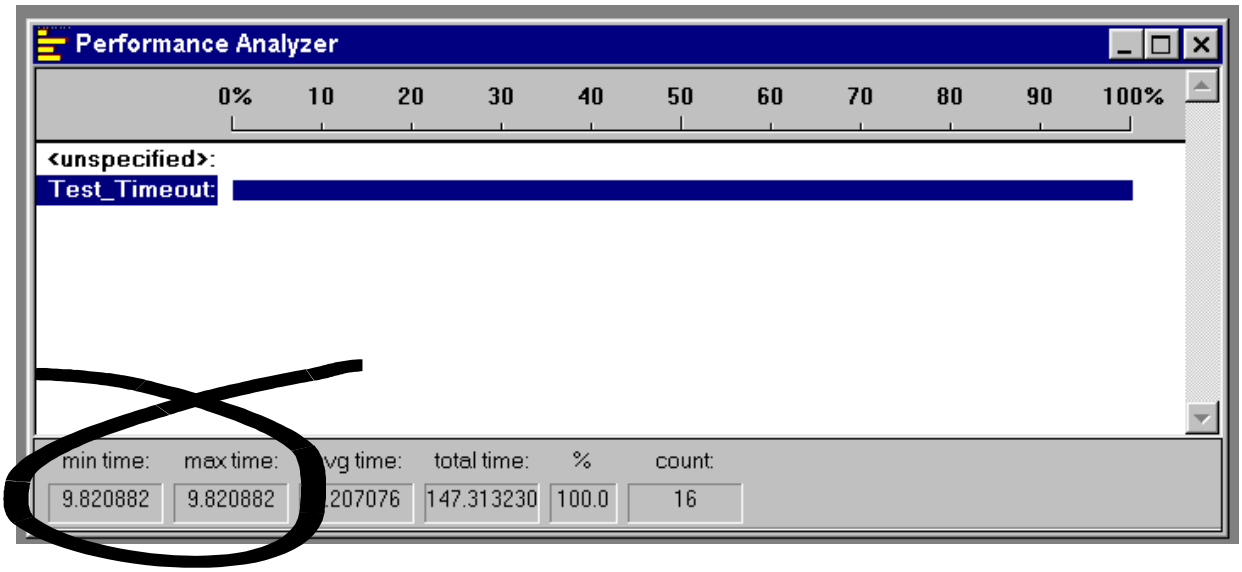
In this seminar, we begin to consider the issues involved in the accurate measurement of time.

These issues are important because many embedded systems must satisfy real-time constraints.



Example: Creating a portable hardware delay

```
/*-------------------------------------------------------------*-

   Main.C (v1.00)

  ------------------------------------------------------------

   Flashing LED with hardware-based delay (T0).

-*-------------------------------------------------------------*/

#include "Main.H"
#include "Port.H"

#include "Delay_T0.h"
#include "LED_Flash.h"

void main(void)
   {
   LED_FLASH_Init();

   while(1)
      {
      /* Change the LED state (OFF to ON or vice versa) */
      LED_FLASH_Change_State();

      /* Delay for *approx* 1000 ms */
      DELAY_T0_Wait(1000);
      }
   }

/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------
-*-------------------------------------------------------------*/
```



Automatic timer reloads

```
/* Preload values for 50 ms delay */
TH0 = 0x3C;         /* Timer 0 initial value (High Byte) */
TL0 = 0xB0;         /* Timer 0 initial value (Low Byte) */

TF0 = 0;            /* Clear overflow flag */
TR0 = 1;            /* Start timer 0 */

while (TF0 == 0); /* Loop until Timer 0 overflows (TF0 == 1) */

TR0 = 0;            /* Stop Timer 0 */
```

For our operating system, we have slightly different requirements:

- We require a long series of interrupts, at precisely-determined intervals.
- We would like to generate these interrupts without imposing a significant load on the CPU.

Timer 2 matches these requirements precisely.

In this case, the timer is reloaded using the contents of the 'capture' registers (note that the names of these registers vary slightly between chip manufacturers):

```
RCAP2H  = 0xFC;   /* Load T2 reload capt. reg. high byte */
RCAP2L  = 0x18;   /* Load T2 reload capt. reg. low byte */
```

This automatic reload facility ensures that the timer keeps generating the required ticks, at precise 1 ms intervals, with very little software load, and without any intervention from the user's program.

The 'automatic' tick interval control is achieved using the C pre-processor, and the information included in the project header file (`Main.H`):

```
/* Oscillator / resonator frequency (in Hz) e.g. (11059200UL) */
#define OSC_FREQ (12000000UL)

/* Number of oscillations per instruction (12, etc) */
...
#define OSC_PER_INST (12)
```

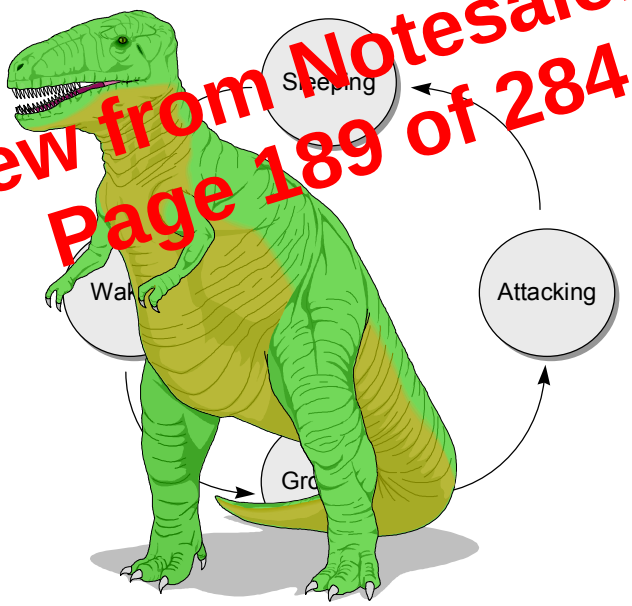
This information is then used to calculate the required timer reload values in `Simple_EOS.C` as follows:

```
   /* Number of timer increments required (max 65536) */
   Inc = ((tLong)TICK_MS * (OSC_FREQ/1000)) / (tLong)OSC_PER_INST;

   /* 16-bit reload value */
   Reload_16 = (tWord) (65536UL - Inc);

   /* 8-bit reload values (High & Low) */
   Reload_08H = (tByte)(Reload_16 / 256);
   Reload_08L = (tByte)(Reload_16 % 256);
...

   TH2     = Reload_08H;   /* Load T2 high byte */
   RCAP2H  = Reload_08H;   /* Load T2 reload capt. reg h byte */
   TL2     = Reload_08L;   /* Load T2 low byte */
   RCAP2L  = Reload_08L;   /* Load T2 reload capt. reg l byte */
```

```
/*-------------------------------------------------------------*-

   Main.c (v1.00)

  ----------------------------------------------------------

   Milk pasteurization example.

-*-------------------------------------------------------------*/

#include "Main.H"
#include "Port.H"
#include "Simple_EOS.H"
#include "Bargraph.H"

#include "Pulse_Count.H"

/* ------------------------------------------------------------ */

void main(void)
   {
   PULSE_COUNT_Init();
   BARGRAPH_Init();

   /* Set up simple EOS (30ms tick interval) */
   sEOS_Init_Timer2(30);

   while(1) /* Super Loop */
      {
      /* Enter idle mode to save power */
      sEOS_Go_To_Sleep();
      }
   }

/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------
-*-------------------------------------------------------------*/
```

```
/*-------------------------------------------------------------*-

   Bargraph.h (v1.00)

  ----------------------------------------------------------

   - See Bargraph.c for details.

-*-------------------------------------------------------------*/

#include "Main.h"

/* ------ Public data type declarations ---------------------- */

typedef tByte tBargraph;

/* ------ Public function prototypes ------------------------- */

void BARGRAPH_Init(void);
void BARGRAPH_Update(void);

/* ------ Public constants ----------------------------------- */

#define BARGRAPH_MAX (9)
#define BARGRAPH_MIN (0)

/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------
-*-------------------------------------------------------------*/
```

Seminar 7:
Multi-State Systems and Function Sequences



Introduction

Two broad categories of multi-state systems:

- **Multi-State (Timed)**
  In a multi-state (timed) system, the transition between states will depend **only** on the passage of time.

  For example, the system might begin in State A, repeatedly executing `FunctionA()`, for ten seconds. It might then move into State B and remain there for 5 seconds, repeatedly executing `FunctionB()`. It might then move back into State A, *ad infinituum*.

  A basic traffic-light control system might follow this pattern.

- **Multi-State (Input / Timed)**
  This is a more common form of system, in which the transition between states (and behaviour in each state) will depend both on the passage of time **and** on system inputs.

  For example, the system might only move between State A and State B if a particular input is received within X seconds of a system output being generated.

  The autopilot system discussed at the start of this seminar might follow this pattern, as might a control system for a washing machine, or an intruder alarm system.

```
/*-----------------------------------------------------------*-
  DINOSAUR_Init()
-*-----------------------------------------------------------*/
void DINOSAUR_Init(void)
   {
   /* Initial dinosaur state */
   Dinosaur_state_G = SLEEPING;
   }

/*-----------------------------------------------------------*-

  DINOSAUR_Update()

  Must be scheduled once per second (from the sEOS ISR).

-*-----------------------------------------------------------*/
void DINOSAUR_Update(void)
   {
   switch (Dinosaur_state_G)
      {
      case SLEEPING:
         {
         /* Call relevant function */
         DINOSAUR_Perform_Sleep_Movements();

         if (++Time_in_state_G == SLEEPING_DURATION)
            {
            Dinosaur_state_G = WAKING;
            Time_in_state_G = 0;
            }

         break;
         }

      case WAKING:
         {
         DINOSAUR_Perform_Waking_Movements();

         if (++Time_in_state_G == WAKING_DURATION)
            {
            Dinosaur_state_G = GROWLING;
            Time_in_state_G = 0;
            }

         break;
         }
```

```
/*-------------------------------------------------------------*/
void DINOSAUR_Perform_Sleep_Movements(void)
    {
    /* Demo only... */
    P1 = (tByte) Dinosaur_state_G;
    P2 = Time_in_state_G;
    }

/*-------------------------------------------------------------*/
void DINOSAUR_Perform_Waking_Movements(void)
    {
    /* Demo only... */
    P1 = (tByte) Dinosaur_state_G;
    P2 = Time_in_state_G;
    }

/*-------------------------------------------------------------*/
void DINOSAUR_Growl(void)
    {
    /* Demo only... */
    P1 = (tByte) Dinosaur_state_G;
    P2 = Time_in_state_G;
    }

/*-------------------------------------------------------------*/
void DINOSAUR_Perform_Attack_Movements(void)
    {
    /* Demo only... */
    P1 = (tByte) Dinosaur_state_G;
    P2 = Time_in_state_G;
    }

/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------
-*-------------------------------------------------------------*/
```

Implementing a Multi-State (Input/Timed) system

- The system will operate in two or more states.
- Each state may be associated with one or more function calls.
- Transitions between states may be controlled by the passage of time, by system inputs or a combination of time and inputs.
- Transitions between states may also involve function calls.

Implementing state timeouts

Consider the following - informal - system requirements:

- The pump should be run for 10 seconds. If, during this time, no liquid is detected in the outflow tank, then the pump should be switched off and 'low water' warning should be sounded. If liquid is detected, the pump should be run for a further 45 seconds, or until the 'high water' sensor is activated (whichever is first).
- After the front door is opened, the correct password must be entered on the control panel within 30 seconds or the alarm will sound.
- The 'down flap' signal will be issued. If, after 50 ms, no flap movement is detected, it should be concluded that the flap hydraulics are damaged. The system should then alert the user and enter manual mode.

To meet this type of requirement, we can do two things:

- Keep track of the time in each system state;
- If the time exceeds a pre-determined error value, then we should move to a different state.

Here is a brief description of the way in which we expect the system to operate:

1. The user selects a wash program (e.g. 'Wool', 'Cotton') on the selector dial.
2. The user presses the 'Start' switch.
3. The door lock is engaged.
4. The water valve is opened to allow water into the wash drum.
5. If the wash program involves detergent, the detergent hatch is opened. When the detergent has been released, the detergent hatch is closed.
6. When the 'full water level' is sensed, the water valve is closed.
7. If the wash program involves warm water the water heater is switched on. When the water reaches the correct temperature, the water heater is switched off.
8. The washer motor is turned on to rotate the drum. The motor then goes through a series of movements, both forward and reverse (at various speeds) to wash the clothes. (The precise set of movements carried out depends on the wash program that the user has selected.) At the end of the wash cycle, the motor is stopped.
9. The pump is switched on to drain the drum. When the drum is empty, the pump is switched off.

```
/* ------ Private variables ---------------------------------- */

static eSystem_state System_state_G;

static tWord Time_in_state_G;

static tByte Program_G;

/* Ten different programs are supported
   Each one may or may not use detergent */
static tByte Detergent_G[10] = {1,1,1,0,0,1,0,1,1,0};

/* Each one may or may not use hot water */
static tByte Hot_Water_G[10] = {1,1,1,0,0,1,0,1,1,0};

/* ---------------------------------------------------------------- */
void WASHER_Init(void)
   {
   System_state_G = INIT;
   }
```

Conclusions

This seminar has discussed the implementation of multi-state (timed) and multi-state (input / timed) systems. Used in conjunction with an operating system like that presented in "Embedded C" Chapter 7, this flexible system architecture is in widespread use in embedded applications.

Overview of this seminar

This seminar will:

- Discuss the RS-232 data communication standard
- Consider how we can use RS-232 to transfer data to and from deskbound PCs (and similar devices).

This can be useful, for example:

- In data acquisition applications.
- In control applications (sending controller parameters).
- For general debugging.

```
void MENU_Perform_Task(char c)
   {
   PC_LINK_IO_Write_Char_To_Buffer(c);  /* Echo the menu option */
   PC_LINK_IO_Write_Char_To_Buffer('\n');

   /* Perform the task */
   switch (c)
      {
      case 'a':
      case 'A':
         {
         Get_Data_From_Port1();
         break;
         }

      case 'b':
      case 'B':
         {
         Get_Data_From_Port2();
         break;
         }
      }
   }

void Get_Data_From_Port1(void)
   {
   tByte Port1 = Data_Port1;
   char String[11] = "\nP1 = XXX\n\n";

   String[6] = CHAR_MAP_G[Port1 / 100];
   String[7] = CHAR_MAP_G[(Port1 / 10) % 10];
   String[8] = CHAR_MAP_G[Port1 % 10];

   PC_LINK_IO_Write_String_To_Buffer(String);
   }

void Get_Data_From_Port2(void)
   {
   tByte Port2 = Data_Port2;
   char String[11] = "\nP2 = XXX\n\n";

   String[6] = CHAR_MAP_G[Port2 / 100];
   String[7] = CHAR_MAP_G[(Port2 / 10) % 10];
   String[8] = CHAR_MAP_G[Port2 % 10];

   PC_LINK_IO_Write_String_To_Buffer(String);
   }
```

Key software components used in this example

This case study uses the following software components:

- Software to control external port pins (to activate the external bell), as introduced in "Embedded C" Chapter 3.
- Switch reading, as discussed in "Embedded C" Chapter 4, to process the inputs from the door and window sensors. Note that - in this simple example (intended for use in the simulator) - no switch debouncing is carried out. This feature can be added, if required, without difficulty.
- The embedded operating system, sEOS, introduced in "Embedded C" Chapter 7.
- A simple 'keypad' library, based on a bank of switches. Note that - to simplify the use of the keypad library in the simulator - we have assumed the presence of only eight keys in the example program (0-7). This final system would probably use at least 10 keys: support for additional keys can be easily added if required.
- The RS-232 library (from "Embedded C" Chapter 9) is used to illustrate the operation of the program. This library would not be necessary in the final system (but it might be useful to retain it, to support system maintenance).

The software

```
/*-------------------------------------------------------------*-

   Port.H (v1.00)

  ----------------------------------------------------

  'Port Header' (see Chap 5) for project INTRUDER (see Chap 10)

-*-------------------------------------------------------------*/

/* ------ Keypad.C ---------------------------------------- */

#define KEYPAD_PORT P2

sbit K0 = KEYPAD_PORT^0;
sbit K1 = KEYPAD_PORT^1;
sbit K2 = KEYPAD_PORT^2;
sbit K3 = KEYPAD_PORT^3;
sbit K4 = KEYPAD_PORT^4;
sbit K5 = KEYPAD_PORT^5;
sbit K6 = KEYPAD_PORT^6;
sbit K7 = KEYPAD_PORT^7;

/* ------ Intruder.C -------------------------------------- */
sbit Sensor_pin = P1^0;
sbit Sounder_pin = P1^7;

/* ------ Lnk_O.C ----------------------------------------- */

/* Pins 3.0 and 3.1 used for RS-232 interface */


/*-------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------
-*-------------------------------------------------------------*/
```

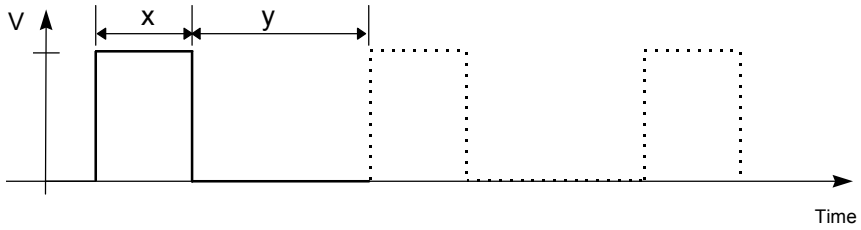
```
bit INTRUDER_Get_Password_G(void)
   {
   signed char Key;
   tByte Password_G_count = 0;
   tByte i;

   /* Update the keypad buffer */
   KEYPAD_Update();

   /* Are there any new data in the keypad buffer? */
   if (KEYPAD_Get_Data_From_Buffer(&Key) == 0)
      {
      /* No new data - password can't be correct */
      return 0;
      }

   /* If we are here, a key has been pressed */

   /* How long since last key was pressed?
      Must be pressed within 50 seconds (assume 50 ms 'tick') */
   if (State_call_count_G > 1000)
      {
      /* More than 5 seconds since last key
         - restart the input process */
      State_call_count_G = 0;
      Position_G = 0;
      }

   if (Position_G == 4)
      {
      PC_LINK_O_Write_Char_To_Buffer('\n');
      }

   PC_LINK_O_Write_Char_To_Buffer(Key);

   Input_G[Position_G] = Key;
```

Conclusions

This case study has illustrated most of the key features of embedded C, as discussed throughout the earlier sessions in this course.

We'll consider a final case study in the next seminar.

Pulse-width modulation



Duty cycle (%) = $\frac{x}{x+y} \times 100$

Period = $x + y$, where x and y are in seconds.

Frequency = $\frac{1}{x+y}$, where x and y are in seconds.

The key point to note is that the average voltage seen by the load is given by the duty cycle multiplied by the load voltage.

See: **"Patterns for Time-Triggered Embedded Systems"**, Chapter 33

Conclusions

That brings us to the end of this course!