University of Portsmouth
Faculty of Technology
Department of Electronic and Computer Engineering

Module: *Principles of DigitalSystems*
Module Code: *B122L*
Module Topic: *Microcontroller Applications*
Lecturer: *Branislav Vuksanovic*

Lecture Notes:

Basics of C Programming for Embedded Systems

This document reviews some general rules of C programming and introduces certain specifics of C programming for 8051 series of microcontrollers. Simple C programs are listed and discussed in details to illustrate the main points. This should provide reader with sufficient knowledge to develop and test other, more complicated C programs for small scale embedded systems employing this microcontrollers.

Content

Introduction to C Programming for Embedded Systems

- most common programming languages for embedded systems are C, BASIC and assembly languages
- C used for embedded systems is slightly different compared to C used for general purpose (under a PC platform)
- programs for embedded systems are usually expected to monitor and control external devices and directly manipulate and use the internal architecture of the processor such as interrupt handling, timers, serial communications and other available features.
- there are many factors to consider when selecting languages for embedded systems
    - Efficiency - Programs must be as short as possible and memory must be used efficiently.
    - Speed - Programs must run as fast as possible.
    - Ease of implementation
    - Maintainability
    - Readability

- C compilers for embedded systems must provide ways to examine and utilise various features of the microcontroller's internal and external architecture; this includes:
    - Interrupt Service Routines
    - Reading from and writing to internal and external memories
    - Bit manipulation
    - Implementation of timers / counters
    - Examination of internal registers

- most embedded C compilers (as well as ordinary C compilers) have been developed supporting the ANSI [American National Standard for Information] but compared to ordinary C they may differ in terms of the outcome of some of the statements
- standard C compiler, communicates with the hardware components via the operating system of the machine but the C compiler for the embedded system must communicate directly with the processor and its components

- For example consider this statement:

```
printf(" C - Programming for 8051\n");
```

In standard C running on a PC platform, the statement causes the string inside the quotation to be displayed on the screen. The same statement in an embedded system causes the string to be transmitted via the serial port pin (i.e TXD) of the microcontroller provided the serial port has been initialized and enabled.

- Another example:

```
c=getch();
```

In standard C running on a PC platform this causes a character to be read from the keyboard on a PC. In an embedded system the instruction causes a character to be read from the serial pin (i.e. RXD) of the microcontroller.

Template for Embedded C Program

```
#include <reg66x.h>
void main(void)
{

// body of the program goes here

}
```

- the first line of the template is the C directive "#include <reg66x.h>"
- this tells the compiler that during compilation, it should look into this file for symbols not defined within the program
- "reg66x.h" file simple defines the internal special function registers and their addresses
- part of "reg66x.h" file is shown below

```
/*-----------------------------------------*/
/* Include file for 8xC66x SFR Definitions   */
/* Copyright Raisonance SA, 1990-2000        */
/*-----------------------------------------*/

/*  BYTE Registers  */
at 0x80 sfr P0   ;
at 0x90 sfr P1   ;
at 0xA0 sfr P2   ;
at 0xB0 sfr P3   ;
at 0xD0 sfr PSW  ;
at 0xE0 sfr ACC  ;
at 0xF0 sfr B    ;
at 0x81 sfr SP   ;
at 0x82 sfr DPL  ;
at 0x83 sfr DPH  ;
at 0x87 sfr PCON ;
at 0x88 sfr TCON ;
at 0x89 sfr TMOD ;
at 0x8A sfr TL0  ;
at 0x8B sfr TL1  ;
at 0x8C sfr TH0  ;
at 0x8D sfr TH1  ;
…
```

In this file, the numerical addresses of different special function registers inside the processor have been defined using symbolic names, e.g. P0 the symbol used for port 0 of the processor is assigned its corresponding numeric address 80 in hexadecimal. Note that in C numbers that are hexadecimal are represented by the 0x.

- the next line in the template declares the beginning of the body of the main part of the program
- the main part of the program is treated as any other function in C program
- every C program should have a main function
- functions are like "procedures" and "subroutines" in other languages
- C function may be written in one of the following formats:
    - it may require some parameters to work on
    - it may return a value that it evaluates or determines
    - it may neither require parameters nor return any value
- if a function requires any parameters, they are placed inside the brackets following the name of the function
- if a function should return a value, it is declared just before the name of the function
- when the word 'void' is used before the function name it indicates that the function does not return any value
- when the word 'void' is used between the brackets it indicates that the function does not require any parameters
- main function declaration:

```
void main(void)
```

therefore, indicates that the main function requires no parameters and that it does not return any value.

- what the function must perform will be placed within the curly brackets following function declaration (your C code)

C Directives

- **#include** is one of many C directives
- it is used to insert the contents of another file into the source code of the current file, as previously explained
- there are two slightly different form of using #include directive:

```
#include < filename >
```

or

```
#include " filename "
```

- the first form (with the angle brackets) will search for the include file in certain locations known to the compiler and it is used to include standard system header files (such are stdlib.h and stdio.h in standard C)
- the second form (with the double quotes) will search for the file in the same directory as the source file and this is used for header files specific to the program and usually written by the programmer
- all directives are preceded with a **"**#" symbol
- another useful directive is a #define directive
- **#define** directive associates a symbolic name with some numerical value or text.
- wherever that symbolic name occurs after the directive the preprocessor will replace it with the specified value or text
- the value of "ON" in the program can be defined as 0xFF throughout the program using:

```
#define ON 0xFF
```

- this approach may be used to define various numerical values in the program using more readable and understandable symbols.
- the advantage of using symbols rather than the actual numerical values is that, if you need to change the value, all you need to do is to change the number that is assigned to the symbol in the define statement rather than changing it within the program which in some cases may be a large program and therefore tedious to do

Example 1

Program to turn the LEDs on port 1 ON (see figure).

```
#include <reg66x.h>
void main(void)
{
      P1 = 0xFF;
}
```

The body of the main function consists of just one statement and that is P1=0xFF. This tells the compiler to send the number 0xFF which is the hexadecimal equivalent of 255 or in binary (11111111) to port 1 which in turn causes the 8 LEDs in port 1 to turn ON.

Note that just like the line in the main body of the program, every line of a C program must end with a semicolon (i.e. ;) except in some special occasions (to be discussed later)

Alternative version of this program using directive #define.

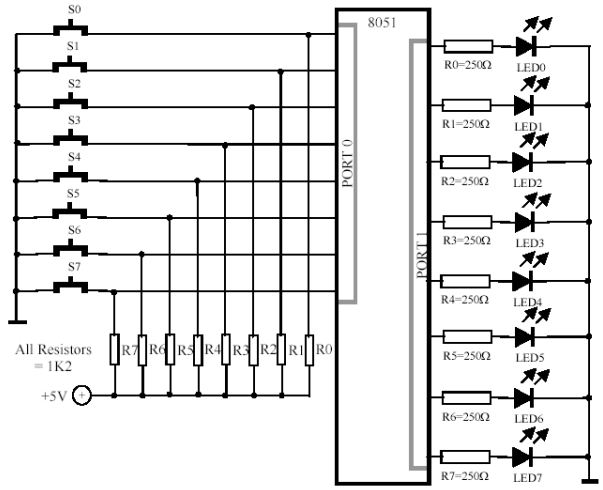
```
#include <reg66x.h>
#define ON 0xFF
void main(void)
{
      P1 = ON;
}
```

In the above example the value of "ON" is defined as 0xFF using:

```
#define ON 0xFF
```

and the statement in the body of the program is then written as:

```
P1 = ON;
```



LEDs and Switches Interfaced to Port 1 and Port 0 of 8051 Microcontroller

Programming Time Delays

- for various reasons it might be necessary to include some sort of time delay routine in most of the embedded system programs
- sophisticated and very accurate techniques using timers/counters in the processor exist to achieve this
- one simple approach (not involving timers) is to let the processor count for a while before it continues
- this can be achieved using a loop in the program where program does not do anything useful except incrementing the loop counter:

```
for(j=0; j<=255; j=j+1)
{
      ;
}
```

Once the loop counter reaches the value of 255 program will exit the loop and continue execution with the first statement following the loop section

- for a longer delays we can use a nested loop structure, i.e. loop within the loop:

```
for(i=0; i<=255; i=i+1)
{
      for(j=0; j<=255; j=j+1)
      {
            ;
      }
}
```

Note that with this structure the program counts to 255 x 255.

Indefinite Loops

- embedded system might be required to continuously execute a section of a program indefinitely
- to achieve this indefinite loop (loop without any exit condition) can be used
- the statement that performs this is:

```
for(;;)
{

}
```

Part of the program to be repeated indefinitely must then be placed in between the curly brackets after the for(;;) statement.

Variables in Embedded C

- variables in C program are expected to change within a program
- variables in C may consist of a single letter or a combination of a number of letters and numbers
- spaces and punctuation are not allowed as part of a variable name
- C is a case sensitive language (therefore I and i are treated as two separate variables)
- in a C program variables must be declared immediately after the curly bracket marking the beginning of a function
- to declare a variable, its type must be defined, it provides information to the compiler of its storage requirement
- some of the more common types supported by C are listed below

Type	Size	Range
unsigned char	1 byte	*0 to 255*
(signed) char	1 byte	*-128 - +127*
unsigned int	2 bytes	*0 - 65535*
(signed) int	2 bytes	*-32768 - +32767*
bit	1 bit	*0 or 1* *(RAM bit-addressable part of memory only)*
sbit	1 bit	*0 or 1* *(SFR bit-addressable part of memory only)*
sfr	8 bit	*RAM addresses 80_h-FF_h only*

- definition of type of a variable in a C program is an important factor in the efficiency of a program
- depending on the type of a variable the compiler reserves memory spaces for that variable
- consider the following guidelines when selecting the type of variables:
    - if speed is important and sign is not important, make every variable unsigned
    - unsigned char is the most common type to use in programming 8051 microcontroller as most registers in the processors are of size 8-bits (i.e one byte)

Example 2

Program to indefinitely flash all LEDs on port 1 at a rate of 1 second.

```
#include <reg66x.h>
void main(void)
{
      unsigned char i,j;
      for(;;)
      {
            P1 = 0xFF; /* Turn All LEDs ON */
            for(i=0; i<=255; i=i+1)
            {
                  for(j=0; j<=255; j=j+1)
                  {
                        ;
                  }
            }
            P1 = 0x00; /* Turn All LEDs OFF */
            for(i=0; i<=255; i=i+1)
            {
                  for(j=0; j<=255; j=j+1)
                  {
                        ;
                  }
            }
      }
}
```

Note that this program contains two identical time delay routines – first one keeps ON state on LEDs for app. 1 second and the second one keeps OFF state on LEDs for the same amount of time. Without those two delays LEDs would switch ON and OFF so rapidly that the whole set would constantly appear as not fully turned ON set of LEDs.

Used delay routine is not very accurate so the delay should only approximately be 1 s.

Note that the first line immediately after the beginning of the main function is used to declare two variables in the program, I and j:

```
unsigned char i,j;
```

C Functions

- when a part of a program must be repeated more than once, a more efficient way of implementing is to call the block to be repeated a name and simple use the name when the block is needed.
- this leads to implementation of C function in the program
- the function (block) must be declared before the main program and defined, normally immediately after the main program is ended
- rules for function declaration are same as for declaration of main function

```
void DELAY (void)
```

Above function declaration informs the compiler that there will be a function called DELAY, requiring no parameters and returning no parameters.
Note the absence of semicolon at the end of function declaration.

- a function is defined in the same way as the main function (an open curly bracket marks the beginning of the function, variables used within the function are then declared in the next line before the body of the function is implemented, the function ends with a closed curly bracket)
- the name of a function must follow the rules for the name of a variable - the function name may not have spaces, or any punctuation
- a use of functions is advisable as functions make programs shorter and readable, a shorter program also requires less space in the memory and therefore better efficiency
- function is called in the main program using function name and appropriate parameters (if any)

Example 3

In this example program from Example 2, to indefinitely flash all LEDs on port 1 at a rate of 1 second is rewritten using function to generate time delay.

```
#include <reg66x.h>
void DELAY(void);
void main(void)
{
      for(;;)
      {
      P1 = 0xFF;        /* Turn All LEDs ON */
      DELAY();          /* Wait for 1 second */
      P1 = 0x00;        /* Turn All LEDs OFF */
      DELAY();
      }
}

void DELAY(void)
{
      unsigned char i, j;
      for(i=0; i<=255; i=i+1)
      {
            for(j=0; j<=255; j=j+1)
            {
            ;
            }
      }
}
```

Other Loops in C

*Two other loops exist in C language - **<do…while>** and **<while>**. Instructions to generate single loop delay using those two loop techniques are given below.*

```
i=0;
do
{
      i=i+1;
} while(i<=255);
```

```
i=0;
while(i<=255)
{
      i=i+1;
}
```

Example 4

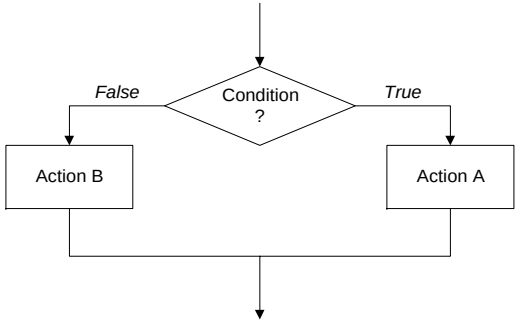
Program from Examples 2 and 3 is rewritten and modified so that LEDs on port 1 flash with 1 s time delay only 10 times.

```
#include <reg66x.h>
void DELAY(void);
void main(void)
{
      unsigned char N;
      N=0;
      do
      {
            P1 = 0xFF;  /* Turn All LEDs ON */
            DELAY();    /* Wait for 1 second */
            P1 = 0x00;  /* Turn All LEDs OFF */
            DELAY();
            N=N+1;
      } while(N<10);
}

void DELAY(void)
{
      unsigned char i,j;
      for(i=0; i<=255; i=i+1)
      {
            for(j=0; j<=255; j=j+1)
            {
                  ;
            }
      }
}
```

Making Decisions in the Program

- an important feature of any embedded system is the ability to test the value of any parameter in the system and based on the outcome of this test take an appropriate action
- the parameter tested must be a program variable and C construct usually employed to perform test is the **<if>** statement
- effect of this statement is illustrated on the flowchart below

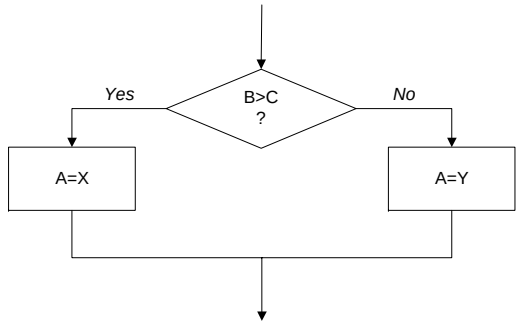


- C instructions to achieve this action are

```
if(condition)
{
      Perform Action A
}
else
{
      Perform Action B
}
```

? Operator

- in the C program operator **<?>** can be used as a short hand version of the 'if' statement discussed above
- flowchart and line of code provided below explain the effect of using this operator




```
A = (B > C) ? X : Y;
```

In other words "if B is greater than C then let A=X otherwise let A=Y".

Example 5

Program to operate the LEDs attached to port 1 as a BCD (Binary Coded Decimal) up counter. A BCD up counter is one, which counts in binary from 0 to 9 and then repeats from 0 again.

```
#include <reg66x.h>
void DELAY(void);
void main(void)
{
      unsigned char N=0;
      for(;;)
      {
            P1 = N;     /* Turn All LEDs ON */
            DELAY();    /* Wait for a while */
            N=N+1;
            if(N>9)
            {
                  N=0;
            }
      }
}

void DELAY(void)
{
      unsigned char i,j;
      for(i=0; i<=255; i=i+1)
      {
            for(j=0; j<=255; j=j+1)
            {
                  ;
            }
      }
}
```

Short Hand Notations

Statement	*Short hand*	*Explanation*
A=A+1	*A++*	*Increment A*
A=A-1	*A--*	*Decrement A*
A=A+B	*A+=B*	*Let A=A+B*
A=A-B	*A-=B*	*Let A=A-B*
*A=A*B*	*A*=B*	*A=A*B*
A=A/B	*A/=B*	*A=A/B*

Logical and Bit-wise Operations

Operation	*In Assembly*	*In C*	*Example in C*
NOT	*CPL A*	*~*	*A=~A;*
AND	*ANL A,#DATA*	*&*	*A= A & DATA*
OR	*ORL A,#DATA*	*¦*	*A= A ¦ DATA*
EX-OR	*XRL A,#DATA*	*^*	*A= A ^ DATA*
Shift Right by n-bits	*RRA*	*>>*	*A=A>>n*
Shift Left by n-bits	*RLA*	*<<*	*A=A<<n*

*The bit-wise AND may be used to perform a **bit mask** operation. This operation may be used to isolate part of a string of bits, or to determine whether a particular bit is 1 or 0. For example, to determine whether the third bit in the 4 bit patter is 1, a bitwise AND is applied to it along with another bit pattern containing 1 in the third bit, and 0 in all other bits, Given a bit pattern 0101 we can bit-wise AND it with **0010**. Result is 0. We therefore know that the third bit in the original pattern was 0. Using bit-wise AND in this manner is called bit masking. The 0 values in the mask pattern **0010** mask the bits that are not of concern, in this case.*

Arrays

- arrays are used when it is necessary to store larger number of data of same type into consecutive memory locations where they can easily be accessed by the program at some later stage
- as any other data in the program array must be declared at the beginning of the main program
- upon reading this declaration compiler will reserve appropriate number of memory locations to store the array elements

```
unsigned char T[20];
```

This array declaration will cause the compiler to reserve 20 consecutive memory locations and call it T. "T" is now the name of that particular array. Note that number inside the square brackets indicate the size of declared array.

Example 6

Two versions of the program that reads the state of the switches on port 0 and operates the LEDs on port 1 accordingly. For example if switch S0 is pressed, program will turn LED0 ON, if S2 and S3 are pressed, then LED 2 and 3 must turn ON and so on.

```
// version 1
#include <reg66x.h>
void main(void)
{
      for(;;)
      {
            P1=P0;
      }
}
```

```
// version 2
#include <reg66x.h>
#define SWITCHES P0
#define LEDS P1
void main(void)
{
      for(;;)
      {
            LEDS = SWITCHES;
      }
}
```

Example 7

This program monitors switch S0 attached to the pin 0 of port 0. When this switch is pressed, it flashes the single LED attached to pin 0 of port 1 ten times. Previously discussed bit masking technique is used to test the value of switch S0.

```
#include <reg66x.h>
#define ON 0x01
#define OFF 0x00
#define mask 0x01       // 00000001

void DELAY(void);
void main(void)
{
   unsigned char S0;
   unsigned char N;


      for(;;)
      {
            S0=P0&mask; // masking  bits
            while(S0)   // wait for key press
            {
                  for(N=0;N<10;N++)
                  {
                        P1=ON;
                        DELAY();
                        P1=OFF;
                        DELAY();
                  }
                  S0=P0&mask;
            }
      }
}
```

```
void DELAY(void)
{
      unsigned char i,j;
      for(i=0; i<=255; i=i+1)
      {
            for(j=0; j<=255; j=j+1)
            {
                  ;
            }
      }
}
```

*Another way to access a single pin on any port of 8051 is to make use of special **sbit** data type. This data type provides access to bit-addressable SFRs and bit-addressable memory space within the processor.*

Line:

```
sbit S0=P0^0;
```

creates an sbit type variable S0 that points to pin 0 of port 0.

Using bit addressing ability of 8051 C compiler, alternative version of the program can be written:

```
#include <reg66x.h>
#define ON 0xFF
#define OFF 0x00

sbit S0 = P0^0;

void DELAY(void);
void main(void)
{
   unsigned char N;


      for(;;)
      {
            while(S0)
            {
                  for(N=0;N<10;N++)
                  {
                        P1=ON;
                        DELAY();
                        P1=OFF;
                        DELAY();
                  }
            }
      }
}
```

Note that because there is no way to indirectly address registers in the 8051, addresses for sbit type variables must be declared outside of any functions within the code.

University of Portsmouth
Faculty of Technology
Department of Electronic and Computer Engineering

Module: *Principles of Digital Systems*
Module Code: *B122L*
Module Topic: *Microcontroller Applications*
Lecturer: *Branislav Vuksanovic*

Lecture Notes:

Programming Timers on 8051

The purpose of this handout is to explain how to use the internal 8051 timers to generate time delays.

Content

Uses of Timers & Counters

The 8051 is equipped with two timers, both of which may be controlled, and configured individually. The 8051 timers have three general functions:

1) Keeping time and/or calculating the amount of time between events (interval timing mode)
2) Counting the events themselves (event counting mode)
3) Generating baud rates for the serial port

The first two uses will be discussed in this handout while the use of timers for baud rate generation will be considered in the handout related to serial ports.

8051 Timers Operation

A timer counts up when incremented by the microcontroller or some external source. When the timer reaches the maximum value and is subsequently incremented, it will reset- or **overflow** -back to 0. This will also set the timer overflow flag in one of timer SFRs.
When a timer is in interval timer mode (as opposed to event counter mode) and correctly configured, it will increment by 1 every machine cycle. If the machine cycle consists of 6 crystal pulses a timer running on the microcontroller with 11.0592 MHz crystal will be incremented 1843200 times per second since:

$$\frac{11.0592\times10^{6}}{6}=1843200$$

This means that if the timer has counted from 0 to 100000 a 0.05425 seconds or 54.25 milliseconds have passed since:

$$\frac{100000}{1843200}=0.05425$$

A reverse calculation can also be made, so if for example, we want to know how many times timer needs to increment during 20 milliseconds we need to perform following calculation:

$$0.02\times1843200=36864$$

It is therefore necessary to have exactly 36864 timer increments to achieve a time period of 20 milliseconds. This is actually a way to measure time using timers on the microcontroller – by counting the number of timer increments during the measured time period. All that is now needed is to learn how to control and initialise timers to provide the needed information. This is done through the proper setting of a number of SFRs related to timer functions and explained in following sections of this document.

Timer Registers

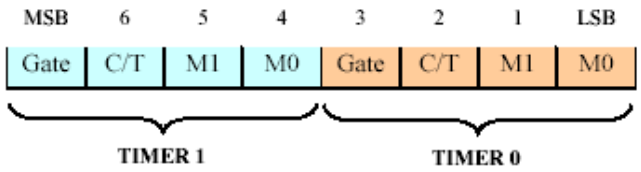
Two 8051 timers are called **Timer0** and **Timer1** and they share two SFRs (**TMOD** and **TCON**) which control the timers, and each timer also has two SFRs dedicated solely to itself (**TH0/TL0** and **TH1/TL1**).
Timer registers names; brief descriptions and addresses are given in the table below.
Timer 0 has two SFRs dedicated exclusively to itself: TH0 and TL0. Those are high and low bytes of the timer. When Timer 0 has a value of 0, both TH0 and TL0 will contain 0. When Timer 0 has the value of for example 1000, TH0 will hold the high byte of the value (3 decimal) and TL0 will contain the low byte of the value (232 decimal). The final value of timer register is obtained by multiplying the high byte value by 256 and adding the low byte value to this value:

$$TH0\times256+TL0=1000$$
$$3\times256+232=1000$$

Timer 1 is identical to Timer 0, but it's SFRs are TH1 and TL1.

SFR Name	Description	SFR Address
TH0	Timer 0 High Byte	8Ch
TL0	Timer 0 Low Byte	8Ah
TH1	Timer 1 High Byte	8Dh
TL1	Timer 1 Low Byte	8Bh
TCON	Timer Control	88h
TMOD	Timer Mode	89h

TMOD SFR



TMOD SFR and its individual bits

The TMOD SFR is used to control the mode of operation of both timers. Each bit of the SFR gives the microcontroller specific information concerning how to run a timer. The high four bits (bits 4 through 7) relate to Timer 1 whereas the low four bits (bits 0 through 3) perform the same functions, but for timer 0.
The individual bits of TMOD and their functions are detailed in the table below.

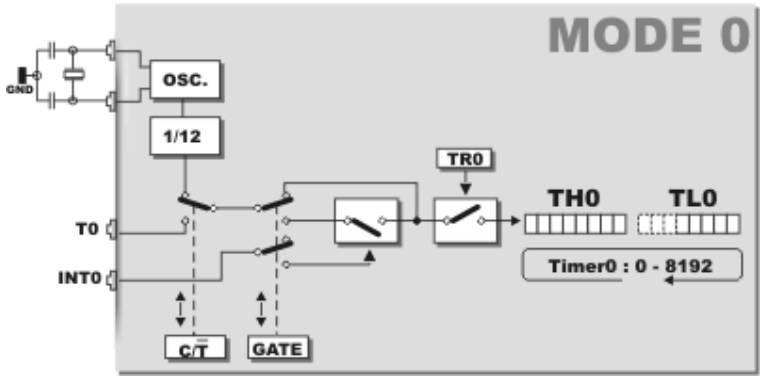
Bit	Name	Explanation of Function
7	**GATE 1**	When this bit is set the timer will only run when INT1 (P3.3) is high. When this bit is clear the timer will run regardless of the state of INT1.
6	**C/T1**	When this bit is set the timer will count events on T1 (P3.5). When this bit is clear the timer will be incremented every machine cycle.
5	**T1M1**	Timer mode bit (see table below)
4	**T1M0**	Timer mode bit (see table below)
3	**GATE 0**	When this bit is set the timer will only run when INT0 (P3.2) is high. When this bit is clear the timer will run regardless of the state of INT1.
2	**C/T0**	When this bit is set the timer will count events on T0 (P3.4). When this bit is clear the timer will be incremented every machine cycle.
1	**T0M1**	Timer mode bit (see table below)
0	**T0M0**	Timer mode bit (see table below)

The modes of operation determined with four bits from the TMOD register are:

TxM1	TxM0	Timer Mode	Mode Description
0	0	0	13-bit Timer
0	1	1	16-bit Timer
1	0	2	8-bit auto-reload
1	1	3	Split timer mode

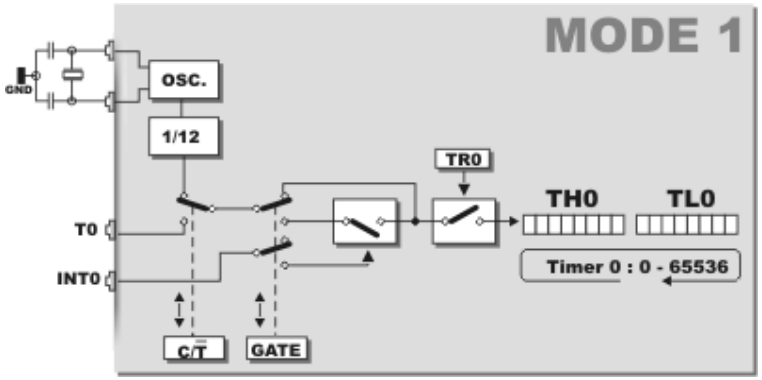
13-bit Time Mode (mode 0)

Timer mode "0" is a 13-bit timer. This mode is used to maintain compatability of 8051 with the older generation microcontroller, the 8048. Generally the 13-bit timer mode is not used in new development. When the timer is in 13-bit mode, TLx will count from 0 to 31. When TLx is incremented from 31, it will "reset" to 0 and increment THx. Thus, effectively, only 13 bits of the two timer bytes are being used: bits 0-4 of TLx and bits 0-7 of THx. This also means, in essence, the timer can only contain 8192 values. If you set a 13-bit timer to 0, it will overflow back to zero 8192 machine cycles later.



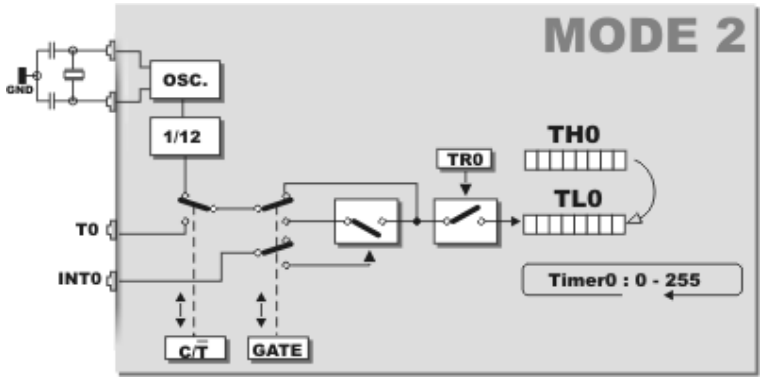
16-bit Time Mode (mode 1)

Timer mode "1" is a 16-bit timer. This is a very commonly used mode. It functions just like 13-bit mode except that all 16 bits are used.
TLx is incremented from 0 to 255. When TLx is incremented from 255, it resets to 0 and causes THx to be incremented by 1. Since this is a full 16-bit timer, the timer may contain up to 65536 distinct values. If you set a 16-bit timer to 0, it will overflow back to 0 after 65,536 machine cycles.



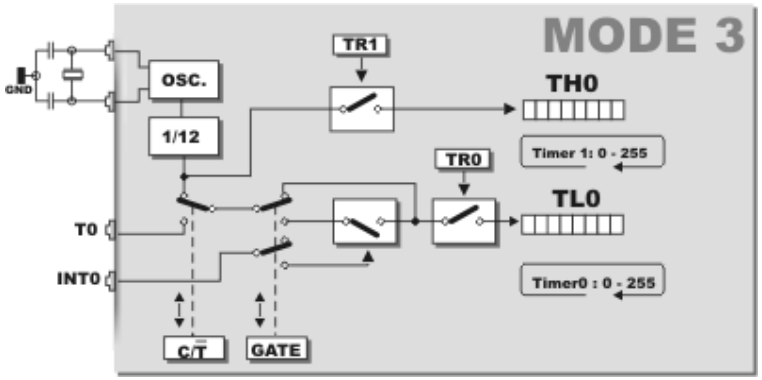
8-bit Time Mode (mode 2)

Timer mode "2" is an 8-bit auto-reload mode. In this mode THx holds the "reload value" and TLx is the timer itself. Thus, TLx starts counting up. When TLx reaches 255 and is subsequently incremented, instead of resetting to 0 (as in the case of modes 0 and 1), it will be reset to the value stored in THx.
In mode 2 THx is always set to a known value and TLx is the SFR that is constantly incremented.
In this mode microcontroller hardware takes care of checking for the timer overfow and reloading of the timer, thus saving some of the processing time.
The auto-reload mode is very commonly used for establishing a baud rate necessary for the serial communications.



Split Timer Mode (mode 3)

Timer mode "3" is a split-timer mode. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. Timer 0 is TL0 and Timer 1 is TH0. Both timers count from 0 to 255 and overflow back to 0. All the bits that are related to Timer 1 will now be tied to TH0.
While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into modes 0, 1 or 2 normally - however, you may not start or stop the real timer 1 since the bits that do that are now linked to TH0. The real timer 1, in this case, will be incremented every machine cycle no matter what.
This mode might be used if there is a need to have two separate timers and, additionally, a baud rate generator. In such case real Timer 1 can be used as a baud rate generator and TH0/TL0 can be used as two separate timers.



TCON SFR

Another register used for the timers control is TCON register.

TCON.7	TCON.6	TCON.5	TCON.4	TCON.3	TCON.2	TCON.1	TCON.0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TCON SFR and its individual bits

It is worth noting here that only upper 4 bits are related to timers operation while the other 4 bits of this register are related to interrupts and will be explained in the handout related to this topic. Therefore. Only the upper 4 bits of this register are listed and explained in the table below. TCON is a “bit-addressable” register; hence an extra column in the table has been added.

Bit	Name	Bit Address	Explanation of Function
7	**TF1**	8Fh	Timer 1 Overflow flag
6	**TR1**	8Eh	Timer 1 Run
5	**TF0**	8Dh	Timer 0 Overflow flag
4	**TR0**	8Ch	Timer 0 Run

Overflow flags TF1 and TF0 are set by the corresponding timers when they overflow. Timer Run bits are used by the program to turn the timer on (when this bit is set to 1) or off (when the bit is cleared).

The benefit of "bit-addressability" of this register is now clear. This has the advantage of setting the high bit of TCON without changing the value of any of the other bits of the SFR. To start or stop a timer no modification of any other value in TCON is necessary.

Once the modes of the operation of the timers are clear and functions of the individual bits of timer initialising registers are known, writing programs to use timers is an easy task.

Procedure of calculating the reload values for timers in order to achieve specific timing delays is summarised in the next paragraph and a number of simple examples using timers for timing or counting external events is also given on the next few pages.

Timer Delay and Timer Reload Value

$$\textit{Timer Delay} = \textit{Delay Value} \times \textit{Timer Clock Cycle Duration}$$

$$\textit{Delay Value} = \textit{how many counts before timer overflows}, \quad \textit{i.e.}$$

$$\textit{Delay Value} = \textit{Maximum Register Count} - \textit{Timer Reload Value}$$

$$\textit{Timer Clock Cycle Duration} = \frac{\textit{number of crystal pulses per machine cycle}}{\textit{crystal frequency}}$$

$$\textit{number of crystal pulses per machine cycle} = \begin{cases} \textbf{6} \text{ for P89C664 model} \\ \textbf{8} \text{ for basic 80C51 model} \end{cases}$$

$$\textit{Maximum Register Count} = \begin{cases} \textbf{65536} \text{ for mode 1} \\ \textbf{256} \text{ for mode 2} \end{cases}$$

$$\textit{Timer Reload Value} = ???$$

Example 1

Calculation of Timer 0 reload value needed to achieve timer delay of 20 ms. Oscillator frequency is 11.0592 MHz.

Delay Value = Timer Delay / Timer Clock Cycle Duration

$$= \frac{20 \times 10^{-3}}{\frac{6}{11.0592 \times 10^{6}}}$$

= 36864 (must be rounded to the nearest integer)

Timer Reload Value = Maximum Register Count - Delay Value

= 65535 – 36864

= 28671

= 0x6FFF

so Timer 0 is loaded with:

TH0 = 0x6F;

TL0 = 0xFF;

Example 2

Function to generate 100 μs delay using timer 0.

Procedure is:

- *Initialise TMOD register*
- *Initialise TL0 and TH0*
- *Start the Timer*
- *Monitor TF0 until it is set*

```
Delay:    MOV   TMOD,#01H   ; initialise TMOD
          MOV   TL0,#47H    ; initialise TL0
          MOV   TL0,#FFH    ; initialise TH0
          SETB  TR0         ; start timer

Wait:     JNB   TF0,Wait    ; wait for TF0
          CLR   TR0         ; stop timer
          CLR   TF0         ; clear TF0
          RET
```

Delay Value $= \frac{100 \times 10^{-3}}{\frac{6}{11.0592 \times 10^{6}}}$ *= 184*

Timer Reload Value = 65535 – 184 = 65351 = 0xFF47

so Timer 0 is loaded with:

TH0 = 0x6F; TL0 = 0xFF;

Example 3

C version of the function from Example 2.

```
void Delay(void)
{
      TMOD = 0x01;
      TL0 = 0x47;
      TH0 = 0xFF;
      TR0 = 1;
      while(!TF0);
      TR0 = 0;
      TF0 = 0;
}
```

Example 4

Program to toggle pin 7 on Port 1 with a time delay of 20 ms.

```
#include <reg66x.h>

#define off 0
#define on 1

sbit pin7 = P1^7; // label pin7 is port 1 pin
7

main()
{
      TMOD = 0x01;
      // timer 0 mode 1,
      // TH0TL0 = 16 bit register

      while(1)
      // keep repeating the following section
      {
            pin7 = on;
            // pin 7 to 5 volts, i.e. logic 1

            // use timer 0 to generate delay
            TH0 = 0x6F; // hex 6F into TH0
            TL0 = 0xFF; // hex FF into TL0
            TR0 = on;   // start timer
            while(!TF0);
            // wait here until TF0 = 1
            TR0 = off;  // stop timer
            TF0 = off;  // clear overflow flag


            pin7 = off;
            // pin 7 to 0 v0lts, i.e. logic 0

            // repeat timer delay
            TH0 = 0x6F; // hex 6F into TH0
            TL0 = 0xFF; // hex FF into TL0
            TR0 = on;   // start timer
            while(!TF0);
            // wait here until TF0 = 1
            TR0 = off;  // stop timer
            TF0 = off;  // clear overflow flag
      }
}
```

Alternative Technique for Timers Loading

Example 5

Load the timer 0 in order to produce 1 kHz square wave (i.e. cycle time of 1000 μs and delay time 500 μs). Oscillator frequency is 11.0592 MHz.

Delay Value = $\frac{500 \times 10^{-6}}{\frac{6}{11.0592 \times 10^{6}}}$ = *922*

Timer Reload Value = *65535 – 922 = 64614 = 0xFC66*

so Timer 0 is loaded with: *TH0 = 0xFC; TL0 = 0x66*

Alternatively if we use: *TH0 = ~(922/256);*

result of integer division 922/256 = 3 will be byte complemented to 0xFC and stored in TH0

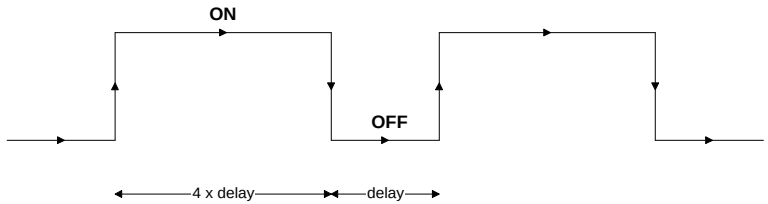
Second line to fill up lower timer 0 register: *TL0 = -(922%256)*

will negate reminder of division 922/256 and store the result in TL0

i.e. *922%256 = 154*
-(922%256) = 256-154 = 102 = 0x66

Example 6

C program to generate 1 kHz square wave from figure below. Square wave should be generated on pin 7 of port 1. Functions are used to generate two delays needed in the program. (delay = 200 μs)




```
// header file containing SFR adresses
#include<reg66x.h>

// to make program more readable:

// define ON and OFF states
#define on 1
#define off 0

// give a name to output pin
sbit pwm = P1^7;

// long and short delay functions
void delay_on();
void delay_off();
```

```
main()
{
      TMOD = 0x01;
      // initialise TMOD for Timer 0 in mode 1

      while(1)    // repeat this
      {
            pwm = on;     // output pin high
            delay_on();   // 800 us delay
            pwm = off;    // output pin low
            delay_off();  // 200 us delay
      }
}

// 800 us delay function
void delay_on()
{
      // loading Timer 0 for longer delay
      TH0 = ~(1475/256);
      TL0 = -(1475%256);
      TR0 = on;     // turn the Timer 0 ON
      while(!TF0);  // wait for timer overflow
      TR0 = off;    // switch the Timer 0 off
      TF0 = off;    // clear the overflow flag
}

// 200 us delay function
void delay_off()
{
      // loading Timer 0 for shorter delay
      TH0 = ~(369/256);
      TL0 = -(369%256);
      TR0 = on;
      while(!TF0);
      TR0 = off;
      TF0 = off;
}
```

University of Portsmouth
Faculty of Technology
Department of Electronic and Computer Engineering

Module: *Principles of Digital Systems*
Module Code: *B122L*
Module Topic: *Microcontroller Applications*
Lecturer: *Branislav Vuksanovic*

Lecture Notes:

Interrupts on 8051

The purpose of this handout is to explain how to handle interrupts on the 8051 series of microcontrollers.

Content

Interrupts

As the name implies, an **interrupt** is some event which interrupts normal program execution.
Program flow is always sequential, being altered only by those instructions which cause program flow to deviate in some way. However, interrupts provide a mechanism to "put on hold" the normal program flow, execute a subroutine (interrupt service routine or interrupt handler), and then resume normal program flow. ISR (interrupt service routine) is only executed when a certain event (interrupt) occurs. The event may be one of the timers "overflowing," receiving a character via the serial port, transmitting a character via the serial port, or one of two "external events." The 8051 may be configured so that when any of these events occur the main program is temporarily suspended and control passed to a special section of code which would execute some function related to the event that occurred. Once complete, control would be returned to the original program so that the main program never even knows it was interrupted.
The ability to interrupt normal program execution when certain events occur makes it much easier and much more efficient to handle certain conditions. Without interrupts a manual check in the main program would have to be performed to establish whether the timers had overflown, whether another character has been received via the serial port, or if some external event had occurred. Besides making the main program longer and more complicated to read and understand, more importantly, such a situation would make program inefficient since "instruction cycles" would be spent checking for events that usually don't happen or happen very infrequently during the program execution.

It was shown in the previous handout how timer flag needs to be checked to detect timer overflow when generating accurate delay in the program. This isn't necessary. Interrupts let us forget about checking for the condition. The microcontroller itself will check for the condition automatically and when the condition is met will jump to a subroutine (called an interrupt handler), execute the code, then return.

Interrupt Events

The 8051 family of microcontrollers can be configured to respond to interrupts caused by the following events:

- Timer 0 Overflow
- Timer 1 Overflow
- Reception/Transmission of Serial Character
- External Event 0
- External Event 1

So when, for example Timer 0 overflows or when a character is successfully sent or received via serial port, the appropriate interrupt handler routines are called and executed. It is necessary to distinguish between various interrupts and to respond to them by executing different ISR. This is accomplished by jumping to a fixed address when a given interrupt occurs. These addresses are usually placed at the bottom of the memory map, in the area of memory so called interrupt vector table (IVT).

Interrupt	Name	ISR Address
External 0	**IE0**	0003h
Timer 0	**TF0**	000Bh
External 1	**IE1**	0013h
Timer 1	**TF1**	001Bh
Serial	**RI/TI**	0023h

From the above IVT it can be seen that whenever Timer 0 overflows (i.e., the TF0 bit is set), the main program will be temporarily suspended and control will jump to 000BH. It is of course necessary to have a code at address 000BH that handles the situation of Timer 0 overflowing.

Enabling Interrupts

By default at power up, all interrupts are disabled. Therefore even if, for example, the TF0 bit is set, the 8051 will not execute the interrupt. User program must specifically tell the 8051 that it wishes to enable interrupts and specifically which interrupts it wishes to enable. Enabling and disabling of the interrupts is done by modifying the IE SFR (Interrupt Enable) (A8h):

IE

MSB							LSB
EA	-	-	ES	ET1	EX1	ET0	EX0

IE SFR and its individual bits

Bit	Name	Bit Address	Explanation of Function
7	**EA**	AFh	Global Interrupt Enable/Disable.
6	-	AEh	Undefined
5	-	ADh	Undefined
4	**ES**	ACh	Enable Serial Interrupt
3	**ET1**	ABh	Enable Timer 1 Interrupt
2	**EX1**	AAh	Enable External 1 Interrupt
1	**ET0**	A9h	Enable Timer 0 Interrupt
0	**EX0**	A8h	Enable External 1 Interrupt

Each of the 8051's interrupts has its own bit in the IE SFR. To enable a given interrupt a corresponding bit needs to be set. To enable Timer 1 Interrupt a bit 3 needs to be set by executing one of two instructions written below:

```
IE = 0x08;   or   ET1 = 1;
```

Once Timer 1 Interrupt is enabled, whenever the TF1 bit is set, the 8051 will automatically put "on hold" the main program and execute the Timer 1 Interrupt Handler at address 001Bh.
However, before Timer 1 Interrupt (or any other interrupt) is truly enabled, bit 7 of IE must also be set. Bit 7, the Global Interrupt Enable/Disable, enables or disables all interrupts simultaneously. If bit 7 is cleared then no interrupts will occur, even if all the other bits of IE are set. Setting bit 7 will enable all the interrupts that have been selected by setting other bits in IE. This is useful in program executing time-critical code where it might be needed to execute code from start to finish without any interrupt getting in the way. To accomplish this bit 7 of IE can simply be cleared (EA = 0;) and then set after the time-critical code is done. So to fully enable the Timer 1 Interrupt the most common approach is to execute the following two instructions:

```
ET1 = 1;
EA = 1;
```

Alternatively, instruction:

```
IE = 0x88;
```

will have the same effect.

Thereafter, the Timer 1 Interrupt Handler at 01Bh will automatically be called whenever the TF1 bit is set (upon Timer 1 overflow).

Interrupt Priorities

When checking for interrupt conditions, 8051 always checks according to predetermined sequence:

1. External 0 Interrupt
2. Timer 0 Interrupt
3. External 1 Interrupt
4. Timer 1 Interrupt
5. Serial Interrupt

This also means that if a Serial Interrupt occurs at the exact same instant that an External 0 Interrupt occurs, the External 0 Interrupt will be serviced first and the Serial Interrupt will be serviced once the External 0 Interrupt has completed.

This priority order in servicing the interrupts on 8051 can be altered. Custom defined priority order offers two levels of interrupt priority: high and low.
If Timer 1 and Serial Interrupt are enabled in the program Timer 1 Interrupt will have a higher priority than the Serial Interrupt, i.e. if two interrupts occur at the same time Timer 1 Interrupt will be serviced first. To change this high priority can be assigned to a Serial Interrupt and low priority to a Timer 1 Interrupt. In this interrupt priority scheme, even if Timer 1 Interrupt is already executing the serial interrupt itself can interrupt the Timer 1 Interrupt. When the serial interrupt ISR is complete, control passes back to Timer 1 Interrupt and finally back to the main program.
Interrupt priorities are controlled by the **IP** SFR (B8h).

IP

MSB							LSB
-	-	-	PS	PT1	PX1	PT0	PX0

IP SFR and its individual bits

Interrupt can be assigned high priority by setting the corresponding bit in this register. Leaving the bit clear, interrupt is assigned the low priority.

Bit	Name	Bit Address	Explanation of Function
7	-	-	Undefined
6	-	-	Undefined
5	-	-	Undefined
4	**PS**	BCh	Serial Interrupt Priority
3	**PT1**	BBh	Timer 1 Interrupt Priority
2	**PX1**	BAh	External 1 Interrupt Priority
1	**PT0**	B9h	Timer 0 Interrupt Priority
0	**PX0**	B8h	External 1 Interrupt Priority

When considering interrupt priorities, the following rules apply:
- Nothing can interrupt a high-priority interrupt-not even another high priority interrupt.
- A high-priority interrupt may interrupt a low-priority interrupt.
- A low-priority interrupt may only occur if no other interrupt is already executing.
- If two interrupts occur at the same time, the interrupt with higher priority will execute first. If both interrupts are of the same priority the interrupt which is serviced first by polling sequence will be executed first.

Interrupt Handling

When an interrupt is triggered, the following actions are taken automatically by the microcontroller:
- instruction currently being executed is finished
- the current content of the Program Counter is stored on the stack, low byte first

- in the case of Timer and External interrupts, the corresponding interrupt flag is cleared.
- ISR address is fetched from IVT (interrupt vector table) and copied over into the Program Counter
- corresponding ISR is executed until the RETI instruction (Return from Interrupt instruction)

An interrupt ends when your program executes the RETI (Return from Interrupt) instruction. When the RETI instruction is executed the following actions are taken by the microcontroller:

- two bytes are popped off the stack into the Program Counter to restore normal program execution.
- interrupt status is restored to its pre-interrupt status.

Timer Interrupts

Timer interrupts occur as a result of Timer 0 or Timer 1 overflow. This sets the corresponding timer flag in TCON register. Flag is cleared automatically by the hardware, so there is no need for the flag clearing instruction in the ISR.

External Interrupts

There is one not so obvious detail, which requires an additional explanation, and it concerns the external interrupts. Namely, if bits IT0 and IT1 (in register TCON) are set, program will be interrupted on change of logic on interrupt pins (P3.2 for External Interrupt 0 and P3.3 for External Interrupt 1) from 1 to 0, i.e. on falling edge of the impulse. If these two bits are cleared, same signal will trigger an interrupt, but in this case it will be continually executed as long as the state on interrupt pins is low.

TCON

MSB							LSB
TF1	TR1	TF0	TR0	**IE1**	**IT1**	**IE0**	**IT0**

Last two unexplained bits of TCON SFR are interrupt flags IE1 and IE0. Similarly to timer flags those two flags are set when corresponding external interrupt occurs. They are cleared by hardware when the CPU vectors to the ISR if the interrupt is transition-activated. If the interrupt is level activated, then IEx flag has to be cleared by the user software

Serial Interrupts

Serial Interrupts are slightly different than the rest of the interrupts. This is due to the fact that there are two interrupt flags: RI and TI (bits 0 and 1 from SCON register). If either flag is set, a serial interrupt is triggered. RI bit is set when a byte is received by the serial port and the TI bit is set when a byte has been sent. This means that when serial interrupt occurs, it may have been triggered because the RI flag was set or because the TI flag was set-or because both flags were set. Thus, interrupt routine must check the status of these flags to determine what action is appropriate. Also, since the 8051 does not automatically clear the RI and TI flags ISR must clear these bits.

Example 1 – Timer Interrupt

```
#include <reg66x.h>

sbit pin7 = P1^7;

void something() interrupt 1 using 2
{
      pin7 = ~pin7;     // complement pin 7
}

main()
{
      while(1)
      {
            TMOD = 0x02;
            // put timer 0 in mode 2
            TL0 = -184;
            TH0 = -184; // reload value
            TR0 = 1;    // start timer 0
            EA = 1;     // global INT enable
            ET0 = 1;    // timer 0 INT enable
            for(;;);    // wait for INT
      }
}
```

Example 2 – External Interrupt

```
#include <reg66x.h>

sbit pin7 = P1^7;
sbit pin6 = P1^6;

void something() interrupt 2 using 2
{
      unsigned int j;
      for(j = 1; j <= 10; j++)      // delay
      pin7 = ~pin7;     // complement pin 7
}

main()
{
      EA = 1;     // global interrupt enable
      EX1 = 1;    // enable P3.3 active low INT
      IT1 = 1;    // = 0 level triggered;
                  // = 1 edge triggered
      while(1)
      {
            unsigned int j;
            for(j = 1; j <=10; j++)
            pin6 = ~pin6;
      }
}
```

University of Portsmouth
Faculty of Technology
Department of Electronic and Computer Engineering

Module: *Principles of Digital Systems*
Module Code: *B122L*
Module Topic: *Microcontroller Applications*
Lecturer: *Branislav Vuksanovic*

Lecture Notes:

Serial Port Operation on 8051

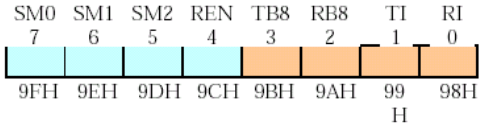
Serial communication is often used either to control or to receive data from an embedded microprocessor. Serial communication is a form of I/O in which the bits of a byte begin transferred appear one after the other in a timed sequence on a single wire. Serial communication has become the standard for intercomputer communication. In this handout the operation of the serial port on the 8051 series of microcontroller is explained.

Content

One of the 8051's features is its integrated *UART*, otherwise known as a serial port. It is very easy way to read and write values to the serial port. If it were not for the integrated serial port, writing a byte to a serial line would be a rather tedious process requiring turning on and off one of the I/O lines in rapid succession to properly "clock out" each individual bit, including start bits, stop bits, and parity bits.
However, on 8051 this is not necessary. Instead, what is needed is to configure the serial port's operation mode and baud rate. Once this is configured, writing to an appropriate SFR will write a value to the serial port or reading the same SFR will read a value from the serial port. The 8051 will automatically signal when it has finished sending the character written to its SFR and will also signal when it has received a byte so that the program can process it further. Programmer does not have to worry about transmission at the bit level, which saves, quite a bit of coding and processing time.

Setting the Serial Port Mode

The first thing to do when using the 8051's integrated serial port is to configure it. This informs the 8051 of how many data bits are to be transmitted or received, the baud rate to be used, and how the baud rate will be determined. **Serial Control Register** (SCON) is the SFR used for this configuration. Bits of this register are listed and their functions explained in the tablebelow.



SCON SFR and its individual bits

Bit	Name	Explanation of Function
7	**SM0**	Serial port mode bit 0
6	**SM1**	Serial port mode bit 1. Explained later.
5	**SM2**	Mutliprocessor Communications Enable
4	**REN**	Receiver Enable. This bit must be set in order to receive characters.
3	**TB8**	Transmit bit 8. The 9th bit to transmit in mode 2 and 3.
2	**RB8**	Receive bit 8. The 9th bit received in mode 2 and 3.
1	**TI**	Transmit Flag. Set when a byte has been completely transmitted.
0	**RI**	Receive Flag. Set when a byte has been completely received.

Additionally, it is necessary to define the function of SM0 and SM1 by an additional table:

SM0	SM1	Serial Mode	Explanation	Baud Rate
0	0	0	8-bit Shift Register	Oscillator / 12
0	1	1	8-bit UART	Set by Timer 1 (*)
1	0	2	9-bit UART	Oscillator / 32 (*)
1	1	3	9-bit UART	Set by Timer 1 (*)

Note: *The baud rate indicated in this table is doubled if PCON.7 (SMOD) is set.*

The first four bits (bits 4 through 7) are configuration bits.

Bits **SM0** and **SM1** set the *serial mode* to a value between 0 and 3, inclusive. The four modes are defined in the chart immediately above. Selecting the Serial Mode selects the mode of operation (8-bit/9-bit, UART or Shift Register) and also determines how the baud rate will be calculated. In modes 0 and 2 the baud rate is fixed based on the oscillator's frequency. In modes 1 and 3 the baud rate is variable based on how often Timer 1 overflows.

The next bit, **SM2**, is a flag for "Multiprocessor communication". Generally, whenever a byte has been received the 8051 will set the "RI" (Receive Interrupt) flag. This lets the program know that a byte has been received and that it needs to be processed. However, when SM2 is set the "RI" flag will only be triggered if the 9th bit received was a "1". That is to say, if SM2 is set and a byte is received whose 9th bit is clear, the RI flag will never be set. This can be useful in certain advanced serial applications. For our applications it is safe to say that this bit needs to be cleared so that the flag is set upon reception of *any* character.

The next bit, **REN**, is "Receiver Enable." This bit is very straightforward: To receive data via the serial port, this bit needs to be set.

The last four bits (bits 0 through 3) are operational bits. They are used when actually sending and receiving data, i.e. they are not used to configure the serial port.

The **TB8** bit is used in modes 2 and 3. In modes 2 and 3, a total of nine data bits are transmitted. The first 8 data bits are the 8 bits of the main value, and the ninth bit is taken from TB8. If TB8 is set and a value is written to the serial port, the data's bits will be written to the serial line followed by a "set" ninth bit. If TB8 is clear the ninth bit will be "clear." The **RB8** also operates in modes 2 and 3 and functions essentially the same way as TB8, but on the reception side. When a byte is received in modes 2 or 3, a total of nine bits are received. In this case, the first eight bits received are the data of the serial byte received and the value of the ninth bit received will be placed in RB8.

TI means "Transmit Interrupt." When a program writes a value to the serial port, a certain amount of time will pass before the individual bits of the byte are "clocked out" the serial port. If the program were to write another byte to the serial port before the first byte was completely output, the data being sent would be overwritten and therefore lost for transmission. Thus, the 8051 lets the program know that it has "clocked out" the last byte by setting the TI bit. When the TI bit is set, the program may assume that the serial port is "free" and ready to send the next byte.

Finally, the **RI** bit means "Receive Interrupt." It functions similarly to the "TI" bit, but it indicates that a byte has been received. Whenever the 8051 has received a complete byte it will trigger the RI bit to let the program know that it needs to read the value quickly, before another byte is read.

Setting the Serial Port Baud Rate

Once the Serial Port Mode has been configured, as explained above, the program must configure the serial port's baud rate. This only applies to Serial Port modes 1 and 3. The Baud Rate is determined based on the oscillator's frequency when in mode 0 and 2. In mode 0, the baud rate is always the oscillator frequency divided by 12. This means if the crystal is 11.059Mhz, mode 0 baud rate will always be 921,583 baud. In mode 2 the baud rate is always the oscillator frequency divided by 64, so a 11.059Mhz crystal speed will yield a baud rate of 172,797. In modes 1 and 3, the baud rate is determined by how frequently timer 1 overflows. The more frequently timer 1 overflows, the higher the baud rate. There are many ways to set timer 1 to overflow at a rate that determines a baud rate, but the most common method is to put timer 1 in 8-bit auto-reload mode (timer mode 2) and set a reload value (TH1) that causes Timer 1 to overflow at a frequency appropriate to generate a baud rate. To determine the value that must be placed in TH1 to generate a given baud rate, we may use the following equation (assuming PCON.7 is clear).

$$TH1 = 256 - \frac{\frac{Crystall}{384}}{Baud}$$

If PCON.7 is set then the baud rate is effectively doubled, thus the equation becomes:

$$TH1 = 256 - \frac{\frac{Crystall}{192}}{Baud}$$

For example, if we have an 11.059Mhz crystal and we want to configure the serial port to 19,200 baud we try plugging it in the first equation:

$$\begin{aligned} TH1 &= 256 - \frac{\frac{Crystall}{384}}{Baud} \\ &= 256 - \frac{\frac{110590}{384}}{19200} \\ &= 256 - \frac{28799}{19200} \\ &= 256 - 1.5 \\ &= 254.5 \end{aligned}$$

Since, timer register TH1 can only be loaded with integer number, if it is set to 254 achieved baud rate is 14,400 and if it is set to 255 achieved baud is 28,800. To achieve exactly 19,200 baud PCON.7 (SMOD) bit needs to be set. This will double the baud rate and utilize the second equation mentioned above:

$$\begin{aligned} TH1 &= 256 - \frac{\frac{Crystall}{192}}{Baud} \\ &= 256 - \frac{\frac{110590}{192}}{19200} \\ &= 256 - \frac{57699}{19200} \\ &= 256 - 3 \\ &= 253 \end{aligned}$$

The calculation results in an integer reload value for TH1. To obtain 19,200 baud with an 11.059MHz crystal we must therefore:

1) Configure Serial Port mode 1 or 3.
2) Configure Timer 1 to timer mode 2 (8-bit autoreload).
3) Set TH1 to 253 to reflect the correct frequency for 19,200 baud.
4) Set PCON.7 (SMOD) to double the baud rate.

Writing to the Serial Port

Once the Serial Port has been properly configured as explained above, the serial port is ready to be used to send data and receive data. To write a byte to the serial port one must simply write the value to the **SBUF** (99h) SFR. Line of C code to send the letter "A" to the serial port is: it could be accomplished as easily as:

```
SBUF = 'A';
```

Upon execution of the above instruction the 8051 will begin transmitting the character via the serial port. Obviously transmission is not instantaneous - it takes a measurable amount of time to transmit. And since the 8051 does not have a serial output buffer we need to be sure that a character is completely transmitted before we try to transmit the next character. The 8051 indicates when it is done transmitting a character by setting the **TI** bit in SCON. When this bit is set the

character has been transmitted and the next character can be send as well. Consider the following code segment:

```
SBUF = 'A'; // write char to SBUF
while(!TI); // wait until done
TI = 0;     // clear interrupt flag
```

The above three instructions will successfully transmit a character and wait for the TI bit to be set before continuing. The program will pause on the "while(!TI)" instruction until the TI bit is set by the 8051 upon successful transmission of the character.

Reading the Serial Port

To read a byte from the serial port the value stored in the **SBUF** (99h) SFR needs to be read after the 8051 has automatically set the **RI** flag in SCON. For example, if the program wants to wait for a character to be received and subsequently read it into the a program variable, the following code segment may be used:

```
while(!RI); // wait for 8051 to set the flag
C = SBUF;   // read the character from the port
RI = 0;
```

The first line of the above code segment waits for the 8051 to set the RI flag; again, the 8051 sets the RI flag automatically when it receives a character via the serial port. So as long as the bit is not set the program waits on the "while(!RI)" instruction. Once the RI bit is set upon character reception the above condition automatically fails and program flow falls through to the next line to read the value received via serial port. Last line of the code segment clears the flag for the next reading of the port.

Example

Following two programs can be used to establish serial communication between two 8051 microcontrollers. First program sends the message, 6 characters long to another 8051 while second program, to be run on another 8051 can receive message of the same length.

```
/* program to send a message, 6 characters
long, to another 8051 via serial channel */

#include <reg66x.h>

char message[6] = {'H','e','l','p',' ','!'};
int letter;

main()
{
      /* initialise serial channel 0 */
      S0CON = 0x40;   /* UART mode 1 */
      TMOD  = 0x20;   /* timer mode 2 */
      TCON  = 0x40;   /* start the timer */
      TH1   = 0xE6;   /* 1200 baud @ 12 MHz */

      /* transmit routine */
      for (letter=0;letter<6;letter++)
      {
            S0BUF = message[letter];
            while (!TI0);
            TI0 = 0;
      }

      /* afterwards wait in endless loop */
      while(1);
}
```

```
/* program to receive a message, 6 characters
long, from another 8051 via serial channel */

#include <reg517.h>

/* reserve space for the message */
char message[6];

int letter;

main()
{
      /* initialise serial channel 0 */
      S0CON = 0x52;
      /* UART mode 1, receive enabled */
      TMOD  = 0x20;   /* timer mode 2 */
      TCON  = 0x40;   /* start the timer */
      TH1   = 0xE6;   /* 1200 baud @ 12 MHz */

      /* receive routine */
      for (letter=0;letter<6;letter++)
      {
            while (!RI0);
            message[letter] = S0BUF;
            RI0 = 0;
      }

      /* afterwards wait in endless loop */
      while(1);/* then wait in endless loop */
}
```