“Programming dsPIC (Digital Signal Controllers) in C”

Programming dsPIC
Microcontrollers in C
All examples presented in this book rely on the programming language C and its application in the mikroC compiler for dsPIC.

Learn how to write your own program, debug it, and use it to start the microcontroller. We have provided plenty of practical examples with necessary connection schematics: temperature sensors, AD and DA converters, LCD and LED displays, relays, communications, and the book is constantly being updated with fresh examples. All code is commented in details to make it easier for beginners. Instruction set, operators, control structures, and other elements of C are thoroughly explained with many examples. Also, the book includes a useful appendix on mikroC for dsPIC development environment: how to install it and how to use it to its full potential.

Authors: Zoran Milivojević, Djordje Šaponjić

Table of Contents

Chapter1: Introduction

1.1 The basic concepts of the programming language C

All examples presented in this book rely on the programming language C and its application in the mikroC compiler for dsPIC. It is thus necessary to introduce some of the concepts, modes of their applications, and their meanings in order to facilitate understanding of the abundant examples contained by the book. The simplest structures of C are used, so the examples can be understood with no need for an analysis of the programming language.

NOTE: In this chapter no complete C language used by the compiler has been presented, but only some of the methods of declaring types and some of the key words often used in the examples. The reader is referred to the help-system accompanying the mikroC compiler for a detailed study of the possibilities of the C language and its application in programming the microcontrollers of the family dsPIC30F.

The memory of a microcontroller of the family dsPIC30F keeps 16-bit (2-bytes) basic data. All other types of data are derived from these. The compiler also supports 1-byte and 4-bytes data. Both integer and floating point variables are supported. This chapter will present the most frequently used types of data.

Table 1-1 presents a review of integer variables supported by the dsPIC mikroC compiler.

Type	Size in bytes	Range
(unsigned) char	1	0 .. 255
signed char	1	- 128 .. 127
(signed) short (int)	1	- 128 .. 127
unsigned short (int)	1	0 .. 255
(signed) int	2	-32768 .. 32767
unsigned (int)	2	0 .. 65535
(signed) long (int)	4	-2147483648 .. 2147483647
unsigned long (int)	4	0 .. 4294967295

Example – declaring integer variable:

```
int i;


long l;
```

Example – declaration of a floating point variable:

The floating point variables are supported by the type of data **float**, **double** and **long double**, 32 bits wide in the memory and of the range (-1.5 * 10^{45} .. +3.4 * 10^{38}).

```
float fnumber;
```

Example – declaration of arrays:

A set of variables of the same type, if indexed, is represented by an array.

```
int vector_one[10]; /* declares an array of 10 integers */


int days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

As can be noticed from the above examples, the indexes in an array can start from any integer value. If within the square brackets only one value is declared, the array has so many members (10 in the above example). In that case the index of the first member iz zero!

Example – declaration of type string:

A series of characters is represented by a type of data called **string**. It, in essense, represents an array of the type **char**, but sometimes it is useful to be considered as text.

```
char txt[6] = "mikro";
```

Example – declaration of pointer:

More difficult to understand, but often used type of data is pointer. It serves for keeping the address in the memory where some essential information is kept. The application of this type of data is of exceptional significance for the module described in **Chapter 11:** DSP Engine.

```
int *ptra;
```

1.2 Keywords

Key words are the concepts in a programming language having special meanings. The names of the variables must not use this set of words. In this chapter 3 key words whose understaning is very important will be treated. More details concerning key words could be found in the help-system accompanying the mikroC compiler for dsPIC.

1.2.1 Key word ASM

Key word **asm** denotes the beginning of an instruction block written in the assembler. The compiler supports the assembler instructions. If an exceptional speed of execution of a part of a code is required, and the user possesses the corresponding knowledge of the microcontroller architecture and assembler instructions, then the critical part of the programe could be written in the assembler (user-optimized parts of the code).

Note: The block of assembler instructions has to start by the key word **asm {** and finish by **}** !

Example – use of the key word asm:

```
asm {

  MOVLW 10  // just a test

  MOVLW _myvar

  MOVLW 0   // just a test

  MOVLW _myvar+1

}
```

When using the key word **asm**, care has to be taken of the available resources. For this reason the use of this key word is not recommended unless it is necessary. The use of the key word **asm** is necessary when using the DSP module which will be discussed in Chapter 11. In this case there is no alternative if the maximum use of the DSP module is targeted.

1.2.2 Key word ABSOLUTE

Key word **absolute** is used in cases when it is required that certain variable is saved at a memory cell which is determined in advance. In majority of cases the use of the key word **absloute** does not make sense. However, its use is sometimes necessary when using the DSP module decribed in **Chapter 11:** DSP Engine.

Example – the use of the key word absolute:

```
int coef[10] absolute 0x0900;


double series[16] absolute 0x1900;
```

In the above example array coef is located at address 0x0900 in the memory. It takes 10x2=20 bytes (10 elements, 2 bytes each), i.e. 20 addresses (10 locations), thus the range of addresses containing the elements of array coef is (0x0900 ... 0x0913). Array series is located at address 0x1900 in the memory. It takes 16x4=64 bytes (16 element, 4 bytes each), i.e. 64 addresses (32 locations) and the range of addresses containing the elements of array series is (0x1900 ... 0x193F).

1.2.3 Key word ORG

The key word **org** is used in situations when it is required that certain method (function) is saved at a preassigned address in the programe memory (the part of the memory where the programe is saved). The use of the key word **org** is necessary when processing the interrupts or traps. The methods (functions) of processing interrupts or traps have to be kept at a precisely specified address.

Example – the use of the key word org:

```
void ADC1_Int() org 0x2A{

  int s;

  IFS0.F11 = 0; //clear AD1IF

  s = ADCBUF0; //fetch sample

}
```

The above example shows the method of instructing the compiler that the given function is saved in the part of the program memory starting from location 0x2A.

1.3 Architecture of the microcontroller

The microcontrollers of the family dsPIC30F are 16-bit modified Harvard architecture processors with an enhanced instruction set. Instructions are 24-bit wide. The working registers are 16-bit wide. There are 16 working registers; the 16th working register (W15) operates as a software stack pointer.

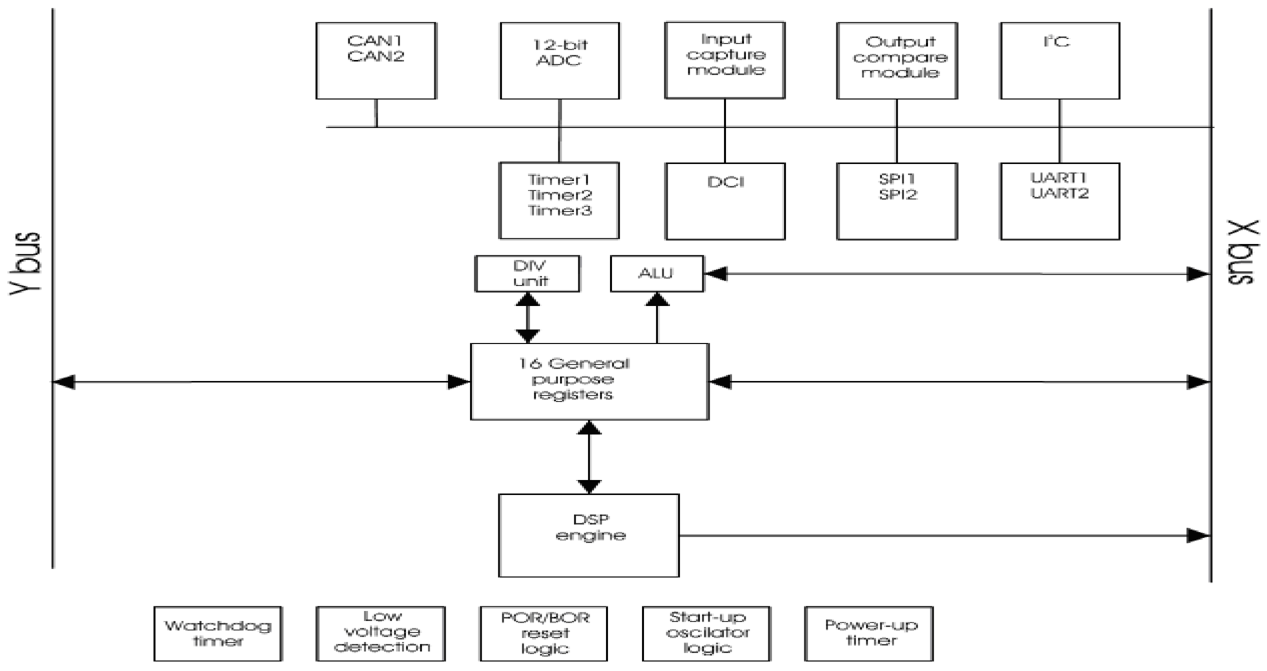


Fig. 1-1 Block diagram of the microcontrollers of the family dsPIC30F

1.4 Abreviations

Abreviation	Full Description

AD	Analogue-to-Digital
DSP	Digital Signal Processing
UART	Universal Asynchronous Receiver Transmitter
I²C	Inter Integrated Circuit
PLL	Phase Locked Loop
PWM	Pulse Width Modulation
MCU	Microprocessor Unit
IVT	Interrupt Vector Table
AIVT	Alternate Interrupt Vector Table
SFR	Special Function Registers
SP	Stack Pointer
PC	Program Counter
IRQ	Interrupt Request
ACC	Accumulator
CAN	Control Area Network
DCI	Data Converter Interface
RTC	Real-Time Clock
GP	General Purpose
PSV	Program Space Visibility
RAM	Random Access Memory
ROM	Read Only Memory
AGU	Address Generator Unit
EA	Effective Address
ALU	Arithmetic Logic Unit
SPI	Serial Peripheral Interface
FIFO	First In – First Out
LIFO	Last In – First Out
PR	Period
SAR	Successive Approximation
SH	Sample and Hold
VREF	Voltage Reference
BUF	Buffer
MUX	Multiplexer
FFT	Fast Fourier Transform
INC	Increment
DEC	Decrement
MS	Master-Slave
SAT	Saturation
IPMI	Intelligent Management Interface
DTMF	Dual Tone Multiple Frequency
FIR	Finite Impulse Response
IIR	Infinite Impulse Response

FRC	Fast RC Oscillator (internal)
FSCM	Fail-Safe Clock Monitor
EXTR	External RESET
BOR	Brown-out Reset
LVD	Low-Voltage Detect

Table 1-2 The abreviations used in the book

Chapter2: Configuration Bits

Introduction

Prior to the start of programming a microcontroller, it is necessary to set the conditions governing the operation of the microcontroller. The governing conditions imply the clock defining the speed of execution of instructions, the source of the clock, the protection against instabilities of the power supply, the protection against irregular execution of the parts of the program (through WATCHDOG timer, to be descibed later), etc.

The configuration of a microcontroller is carried out by the four registers:

Register	Name	Address
FOSC	Oscillator Configuration Register	0xF80000
FWDT	Watchdog Timer Configuration Register	0xF80002
FBORPOR	BOR and POR (Voltage Protection) Configuration Register	0xF80004
FGS	General Code Segment Configuration Register	0xF8000A

These registers are in the programming memory in the upper half of the memory. Details concerning the programming memory can be found in Chapter 8 of the book. It is important to know that the locations in the programming memory are 24-bit wide.

Attention!
The access to the configuration registers is enabled only during configuration of the microcontroller. Any attempt to access the configuration registers in terms of the programming code will result in the microcontroller reset.

2.1 Oscillator configuration register (FOSC)

The oscillator configuration register is used for adjusting the clock of the microcontroller. The clock is of direct influence on the speed of execution of instructions. It is thus essential that the clock is selected correctly. It is also essential to select the oscillator source. The source can be the internal RC oscillator (FRC – internal fast RC oscillator, LPRC – internal low power RC oscillator) or external oscillator (ERC – external RC oscillator, XT – external quartz).

The internal RC oscillator is insufficiently accurate for the majority of microcontroller applications. It is its dependence on the supply voltage and temperature that makes it unacceptable. Besides the internal RC oscillator, it is possible to use an external crystal resonator as a source of the clock. Its frequency could vary from 200kHz to 25MHz. The trird option is to use an external clock (external oscillator or the clock from another instrument). If an external source of the clock or a crystal resonator is used, it is possible to use the internal PLL and increase the internal clock (the clock of the execution of instructions) by factor 4, 8, or 16.

The internal clock, of period TCY, controlling the execution of instructions, is obtained when the clock (from the PLL or directly) is divided by 4. Let a 10MHz crystal resonator and PLL 4x are selected. This means that the internal clock is 4x10MHz =40MHz. This value is divied by 4 to obtain 10MHz (100ns period). This implies that one instruction is executed in 100ns, the internal clock period. In the majority of processes it is very important to know the times of the execution of parts of the program. For these calculations it is necessary to know the internal clock period of the microcontroller.

The oscillator configuration register: selects the source of the clock, controls the activation and mode of thePLL, enables the permission to switch to another source of the clock during operation, and controls monitoring of the clock.

The family of dsPIC30F microcontrollers has the ability that during operation, if it becomes necessary (e.g. one source of the clock fails), can switch to another source of the clock. This ability provides a whole spectrum of applications of the dsPIC30F family of microcontrollers. It is possible to provide an additional stability of the clock, switch to a faster source of the clock within the parts of the program that have to be executed faster, swith to the low power operation, etc. The structure of the oscillator configuration register is given in Table 2-1.

name	**ADR**	**23-16**	**15-14**	**13**	**12**	**11**	**10**	**9-8**	**7**	**6**	**5**	**4**	**3-0**
FOSC	0xF80000	-	FCKSM<1:0>	-	-	-	-	FOS<1:0>	-	-	-	-	FPR<3:0>

Table 2-1 Description of the oscillator configuration register FOSC

```
FCKSM <1:0> - Clock switching selection
              Clock monitoring selection
       1x - Switching between different sources of the clock is disabled.
            Clock monitoring is disabled.
       01 - Switching between different sources of the clock is enabled.
            Clock monitoring is disabled.
       00 - Switching between different sources of the clock is enabled.
            Clock monitoring is enabled.

FOS <1:0> - Definition of the clock source.
     11 - Primary oscillator selected by FPR<3:0>
     10 -Internal low power RC oscillator
     01 -Internal fast RC oscillator
     00 - Low power 32kHz internal oscillator (timer 1)

FPR <3:0> - Selection of the mode of generating the internal clock
     1111 - EC PLL 16x - External clock mode with 16x PLL enabled.
            External clock at  OSC1, OSC2 pin is I/O
     1110 – EC PLL 8x - External clock mode with 8x PLL enabled.
            External clock at  OSC1, OSC2 pin is I/O
     1101 – EC PLL 4x - External clock mode with 4x PLL enabled.
            External clock at  OSC1, OSC2 pin is I/O
     1100 – ECIO – External clock mode without PLL.
```

```
            External clock at OSC1, OSC2 pin I/O
     1011 – EC – External clock mode. OSC2 pin is system clock output
(Fosc/4)
     1010 – Reserved
     1001 – ERC – External RC oscillator mode. OSC2 pin is system clock
output (Fosc/4)
     1000 – ERCIO – External RC oscillator mode. OSC2 pin is I/O
     0111 – XT PLL 16x – XT crystal oscillator mode with 16xPLL enabled. 4
– 10MHz crystal
     0110 – XT PLL 8x – XT crystal oscillator mode with 8xPLL enabled. 4 –
10MHz crystal
     0101 – XT PLL 4x - XT crystal oscillator mode with 46xPLL enabled. 4 –
10MHz crystal
     0100 – XT -  XT crystal oscillator mode (4 – 10MHz crystal)
     001x - HS - HS crystal oscillator mode (10 – 25MHz crystal)
     000x – XTL - XTL crystal oscillator mode (200kHz – 4MHz crystal)
```

2.2 Configuration register of the watchdog timer (FWDT)

The configuration register of the watchdog timer serves for swithing on and off of the watchdog timer and for the adjustment (preseting) of its operation. The preset value is a number dividing the internal RC clock to obtain, in this case, the clock of the watchdog timer. The watchdog timer is independent of the internal clock which governs the execution of instructions. In this way an operative timer is obtained even if the internal clock fails. The frequency of the RC oscillator for the watchdog timer is 512kHz. After the initial division by 4 one obtains the 128kHz which is the basic clock of the watchdog timer.

The basic clock 128kHz of the watchdog timer is divided by two progammable values. The first is **prescaler A** and the **second prescaler B**. After division, the internal clock of the watchdog timer is obtained. With this clock an 8-bit register is incremented. When the value in the register reaches the maximum of 255, the watchdog timer resets the microcontroller or switches the microcontroller from inactive to active state. If the microcontroller is in the inactive (SLEEP) state, the watchdog swithes it into the active state and the execution of the current program continues. If the microcontroller is in the active mode, i.e. the execution of the current program is on, the watchdog timer will, upon the expiration of the defined time, reset the miricontroller.

The time interval when the wacthdog sends the reset/wake-up signal is calculated as follows. The period of the basic clock is 1/(128 kHz) = 7.8125µs. This value is multiplied by the prescaler A and prescaler B values. This gives the clock period of the watchdog timer. This period multiplied by 255 gives the maximum value the register will be incremented. This value is the time the watchdog timer will wait before it generates the reset/wake-up signal of the microcontroller. The waiting times, in milliseconds, depending on the values of prescaler A and prescaler B, are given in table 2-2.

Prescaler B value	**Prescaler A value**			
	1	8	64	512
1	2	16	128	1024
2	4	32	256	2048
3	6	48	384	3072
4	8	64	412	4096
5	10	80	640	5120
6	12	96	768	6144
7	14	112	896	7168
8	16	128	1024	8192

9	18	144	1152	9216
10	20	160	1280	10240
11	22	176	1408	11264
12	24	192	1536	12288
13	26	208	1664	13312
14	28	224	1792	14336
15	30	240	1920	15360
16	32	256	2048	16384

Table 2-2 Waiting times of the watchdog timer

Example:

Let the execution of a part of the program, depending on an external signal, should not last longer than 100ms. In this case the watchdog timer can be activated with the period of 100ms. If it occus that the signal is 100ms late, the microcontroller is reset and the execution of the program starts from the beginning. The watchdog timer should be configured to perform a given function. It is impossible to achieve the waitng time of 100ms, so the period of 112ms (see table 2-2) will be used as the shortest waiting time after 100ms. This is accomplshed with the values of prescaler A = 8 and prerscaler B = 7.

If the program is to function correctly, it is necessary to reset at times the value in the watchdog register. This is achieved by the assembler instruction CLRWDT.
The watchdog timer is configured by the FWDT register. The structure of the configuration register of the watchdog timer is shown in table 2-3.

name	ADR	23-16	15	14	13	12	11	10	9	8	7	6	5-4	3-0
FWDT	0xF0002	-	FWDTEN	-	-	-	-	-	-	-	-	-	FWPSA<1:0>	FPR<3:0>

Table 2-3 The configuration register od the watchdog timer FWDT

```
FWDTEN – Watchdog enable configuration bit
         1 – Watchdog enabled; LPRC osillator can not be disabled
         0 – Watchdog disabled

FWPSA <1:0> - Selected value of prescaler A
       11 = 1:512
       10 = 1:64
       01 = 1:8
       00 = 1:1

FWPSB <1:0> - Selected value of prescaler B
       1111 = 1:16
       1110 = 1:15
       ....
       ....
       ....
       0001 = 1:2
       0000 = 1:1
```

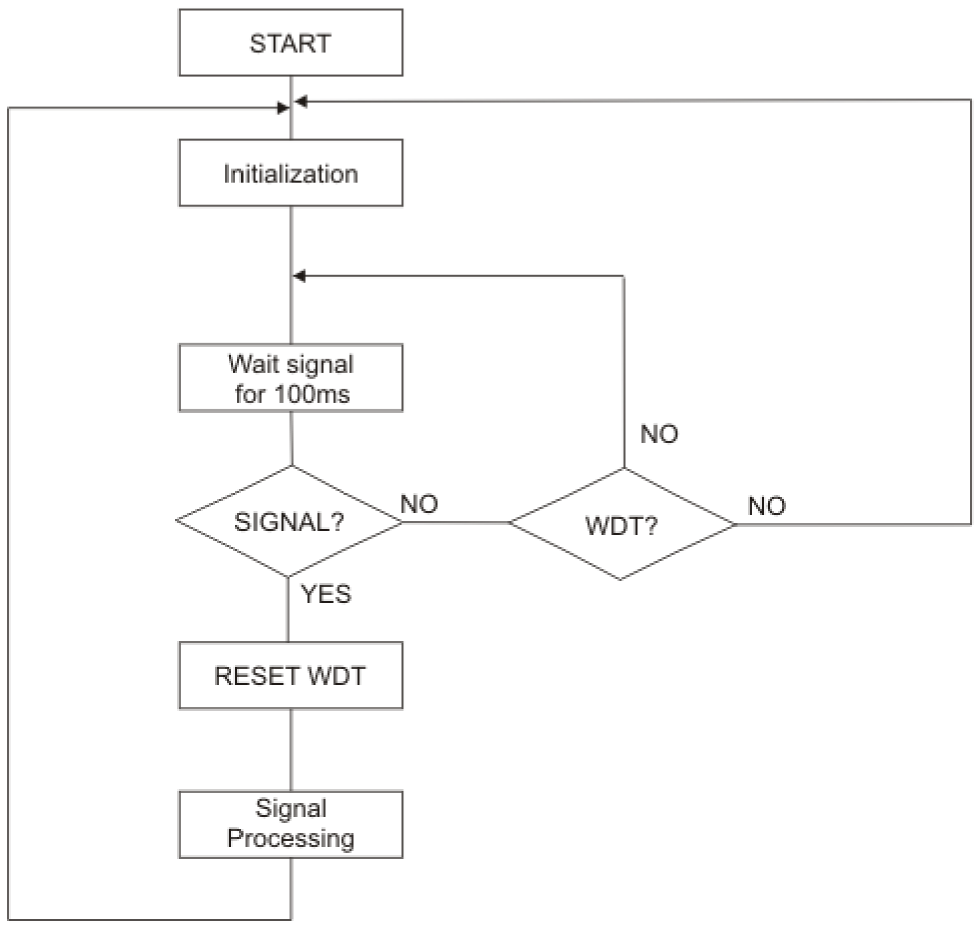


Fig. 2-1 The algorithm of the watchdog timer

2.3 Voltage protection configuration register (FBORPOR)

The configuration register of the voltage protection serves for defining the minimum value of the supply voltage, switching on/off the circuit for reseting the microcontroller if the supply voltage falls bellow the specified value, and for adjustment of the PWM output.

The proper execution of programs requires a godd stability of the power supply. The family of microcontrollers dsPIC30F has the ability of defining a minimum power supply voltage ensuring proper functioning of the device. If the supply voltage falls below this limit, the internal circuit will reset the microcontroller, wait until the supply voltage returns above the limit, and after the specified power-up delay time activate the microcontroller starting the execution of the program for the beginning.

The adjustment of the PWM output, more elaborated in Chapter 6 Output Compare Module, makes also the contents of this register. It is possible to select the states of PWM pins at a device reset (high impedance or output) and the active signal polarity for the PWM pins. The polarity for the high side and low side PWM pins may be selected independently.

name	**ADR**	**23-16**	**15**	**14-11**	**10**	**9**
FBORPOR	0x80004	-	MCLREN	-	PWMPIN	HPOL

Table 2-4. The voltage protection configuration register FBORPOR

8	**7**	**6**	**5-4**	**3-2**	**1-0**
LPOL	BOREN	-	BORV<1:0>	-	FPWRT<1:0>

(Table 2-4. continued)

```
MCLREN – Pin function enable bit
         below the specified value
         1 - Pin function enabled (default)
         0 - Pin is disabled

PWMPIN – The state of PWM at device reset
         1 - PWM pin is in the state of high impedance at device reset
         0 - PWM pin is configured as output at device reset

HPOL – High side polarity bit
       1 - High side output pin has active-high output polarity
       0 - High side output pin has active-low output polarity

LPOL - Low side polarity bit
       1 - Low side output pin has active-high output polarity
       0 - Low side output pin has active-low output polarity

BOEN – Enable bit
       1 - PBOR enabled
       0 - PBOR disabled

BORV <1:0> - Minimum supply voltage select bits
      11 - 2.0V
      10 - 2.7V
      01 - 4.2V
      00 - 4.5V

FPWRT <1:0> - Power-on reset timer value selection bits
       11 - 64ms
       10 - 16ms
       01 - 4ms
       00 - 0ms
```

2.4 Program memory protection configuration register (FGS)

The program memory protection configuration register is used to code protect or write protect the program memory space. It includes all user program memory with the exception of the interrupt vector table space.

If the program memory is code protected, the device program memory can not be read from the device using in-circuit serial programming or the device programr. Further code cannot be programd into the device without first erasing the entire general code segment.

The program memory protection register uses only two configuration bits, the others are unimplemented. The two configuration bits must be programd as a group, i.e. it is not possible to change only one of them.

If the code protect and write protect of the program memory are enabled, a full chip erase must be performed to change the state of either bit.
The structure of the program memory protection configuration register is shown in table 2-5.

name	ADR	23-16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FGS	0x8000A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 2-5. Program memory protection configuration register FGS

```
GCP - Program memory code protect enable
      1 - Program memory is not code protected
      0 - Program memory is code protected

GWRP - Program memory write protect enable
       1 – Program memory is not write protected
       0 - Program memory is write protected
```

Chapter3: Interrupts and Traps

3.1 Interrupts

Interrupts are unexpected events in a sequence of execution of instructions causing an interruption of the normal programme flow. The causes of interrupts can be different. For the family of dsPIC30F microcotrollers, in the transducer or general purpose applications, 41 interrupt sources and 4 types of traps, ranked according to the priority schematic, have been specified.

For each specified interrupt or trap the microcontroller (MCU) has a clearly defined further programme sequence specified by the interrupt vector table (IVT). The IVT contains the initial addresses of interrupt routines for each interrupt source or trap. IVT is located in the programme memory starting at location 0x000004 up to location 0x00007E.

In addition to the interrupt vestor table, in the family of dsPIC30F microcontrollers, the alterante interrupt vector table (AIVT) is also specified. The purpose of the AIVT is to enable the realization of alternate interrupt routines. The AIVT is also located in the programme memory starting at location 0x000084 up to location 0x0000FE.

On the basis of the priorities specified by the special function registers, the interrupt controller is responsible for the preparation and preprocessing of interrupts:

- **IFS0**<15:0>, **IFS1**<15:0>, and **IFS2**<15:0> are the registers containing all the interrupt request flags. Each source of interrupt has a status bit set by the respective peripherals or external signals. These flags are cleared by the user software.
- **IEC0**<15:0>, **IEC1**<15:0>, and **IEC2**<15:0> are the registers containing all the interrupt enable control bits. These control bits are used to individually enable interrupts from the peripherals or external signals.
- **IPC0**<15:0>, **IPC1**<15:0>, ... **IPC10**<7:0> are the registers used to set the interrupt priority level for each of the 41 sources of interrupt.
- **IPL**<3:0> are the bits containing the current CPU priority level. **IPL**<3> bit is located in the register **CORCON** and the remaining three bits **IPL**<2:0> are in the STATUS register (SR) of the microcontroller.
- **INTCON1**<15:0> and **INTCON2**<15:0> are the registers containing global interrupt control functions. **INTCON1** contains the control and status bits for the processor trap sources; **INTCON2** controls the external interrupt requests behaviour and the use of the alternate vector table by setting the **ALTIVT** bit (**INTCON2**<15:0>).

During processor initialization it is necessary to enable interrupts which will be used and assign the corresponding priority levels.

NOTE: Interrupt request flags are set in the IFSx registers irrespective of the interrupt enable bits in the IECx registers. When the conditions for an interrupt to occur are met, the corresponding bits in the IFS are set. Before an interrupt is allowed, the user software has to clear the corresponding IFSx bits because the interrupt controller, upon detecting an interrupt request will not be able to process new interrupt involving these IFSx bits.

Interrupt nesting is enabled by the **NSTDIS** bit of the control register **INTCON1**. Nesting of interrupt routines may be optionally disabled by setting this bit. This may be significant if it is necessary to carry out a part of the programme without an interrupt that could change the state which is the basis for this part of the programme or if the interrupt routine changes some variable which are of significance for further execution of the programme.

The priority level of each interrupt is assigned by setting the interrupt priority bits IP<2:0> for each source of interrupt. The bits IP<2:0> are the least significant 3 bits of each nibble (4 bits) within the IPCx register. Bit no. 3 of each nibble iz always zero. The user can assign 7 priority leveles, from 1to 7. Level 7 is the highest and level 1 the lowest priority level for the maskable interrupts, i.e. for the interrupts that could be enabled by the control bits of IECx.

Attention!
If an interrupt is assigned priority level 0, it is the same as if no interrupt is allowed by the bits set in IECx.

Natural order priority is specified by the position of an interrupt in the vector table (IVT). It is used only to resolve conflicts between simultaneous pending interrupts with the same user assigned priority level. Then, the interrupt of the higher natural level of priority is executed first. As an example, table 3-1 shows for the microcontroller dsPIC30F4013 the interrupt vector table (IVT) with all sources of interrupts, interrupt number in the vector table, and the number which defines the natural order priority.

INT Num	Vector Num	IVT Address	AIVT Address	Interrupt Source
Highest Natural Order Priority				
0	8	0x000014	0x000094	INT0 - External Interrupt 0
1	9	0x000016	0x000096	IC1 - Input Capture 1
2	10	0x000018	0x000098	OC1 - Output Compare 1
3	11	0x00001A	0x00009A	T1 - Timer 1
4	12	0x00001C	0x00009C	IC2 - Input Capture 2
5	13	0x00001E	0x00009E	OC2 - Output Compare 2
6	14	0x000020	0x0000A0	T2 - Timer 2
7	15	0x000022	0x0000A2	T3 - Timer 3
8	16	0x000024	0x0000A4	SPI1
9	17	0x000026	0x0000A6	U1RX - UART1 Receiver
10	18	0x000028	0x0000A8	U1TX - UART1 Transmitter
11	19	0x00002A	0x0000AA	ADC - ADC Convert Done
12	20	0x00002C	0x0000AC	NVM - NVM Write Complete

13	21	0x00002E	0x0000AE	SI2C - I2C Slave Interrupt
14	22	0x000030	0x0000B0	MI2C - I2C Master Interrupt
15	23	0x000032	0x0000B2	Input Change Interrupt
16	24	0x000034	0x0000B4	INT1 - External Interrupt 1
17	25	0x000036	0x0000B6	IC7 - Input Capture 7
18	26	0x000038	0x0000B8	IC8 - Input Capture 8
19	27	0x00003A	0x0000BA	OC3 - Output Compare 3
20	28	0x00003C	0x0000BC	OC3 - Output Compare 4
21	29	0x00003E	0x0000BE	T4 - Timer 4
22	30	0x000040	0x0000C0	T5 - Timer 5
23	31	0x000042	0x0000C2	INT2 - External Interrupt 2
24	32	0x000044	0x0000C4	U2RX - UART2 Receiver
25	33	0x000046	0x0000C6	U2TX - UART2 Transmitter
26	34	0x000048	0x0000C8	Reserved
27	35	0x00004A	0x0000CA	C1 - Combined IRQ for CAN1
28-40	36-48	0x00004C – 0x000064	0x0000CC – 0x0000E4	Reserved
41	49	0x000066	0x0000E6	DCI - CODEC Transfer Done
42	50	0x000068	0x0000E8	LVD - Low Voltage Detect
43-53	51-61	0x00006A – 0x00007E	0x0000EA – 0x0000FE	Reserved
Lowest Natural Order Priority				

Table 3-1. Interrupt vector table of microcontroller dsPIC30F4013

Example:

The example shows how dsPIC reacts to a rising signal edge at the pin RF6(INT0). For each rising edge the value at port D is incremented by 1.

NOTE: Pins that can be used for external interrupts are model dependent. This example is made for dsPIC30F6014A

```
void IntDetection() org 0x0014{ //Interrupt on INT0

  LATD++;

  IFS0.F0 = 0;                //interrupt flag cleared

}
```

```
void main(){

  TRISD = 0;      //portd is output

  TRISF = 0xFFFF; //portf is input

  IFS0 = 0;       //interrupt flag cleared

  IEC0 = 1;       //Interrupt is set on a rising edge at INT0 (RF6)

  while(1) asm nop;

}
```

Let us summarize the algorithm how dsPIC processes interrupts in this example. Two ports are used, PORTD as the output to display the number of interrupt events and PORTF as the input; this means that an interrupt will occur when at INT0 (RF6) logic 0 changes to logic 1. In the register IEC0 the least significant bit (IEC0.0) is set for allowing reaction to interrupt INT0. The meanings of other bits are shown in table 3-7. When an interrupt occurs, the function **IntDetection** is called. How does dsPIC “know” to call exactly this function? By instruction **org** in the interrupt vector table (see table 3-1) at the memory location 0x000014 is written the function **IntDetection**.

What does dsPIC do when an interrupt occurs, i.e. when at RF6 logic 1 appears after logic 0? At first, it writes logic 1 in the least significant bit of the register IFS0. Then, it tests if the interrupt INT0 is enabled (the least significant bit of IEC0). If yes, it reads from the interrupt vector table which part of the programme should be executed. mikroC compiler will, at position 0x000014, write the code where function **IntDetection** starts and dsPIC will execute this function and return to the main program.

Two operations are carried out in the interrupt function. At first, the interrupt flag is cleared **(dsPIC does not do that automatically, but leaves this to the user software)**. Then, the value at port D is incremented by 1 with **LATD++**.

What would happen if the interrupt flag in the IFS register was not cleared? If this line is omitted, dsPIC will wait until first interrupt, call and execute the function **IntDetection**. What is the difference? The register IFS0 will still indicate that an interrupt occured. As soon as the microcontroller returns to the main programme and executes one instruction, it would understand that an interrupt occured again and the **IntDetection** function would be called again. This means that **IntDetection** function would be executed after each instruction of the main programme. The reader is encouraged to erase the line **`IFS0.F0:=0`**; and see what happens.

What is the purpose of while(1) asm nop;? When dsPIC comes to the end of a programme, it starts from the beginning as if reset. However, at the end of the programme the compiler inserts an endless loop to prevent this of happening. This has been inserted here to make the reader aware of the existence of this loop and that reseting dsPIC is not possible.

3.2 Traps

Traps can be considered as interrupts that could not be masked by setting interrupt control bits in the interapt enable register IECx. Traps are intended to provide the user a means to correct erroneous operation during debug and development of applications.

NOTE: If the user does not intend to use corrective action in the event of a trap error condition, the interrupt vector table is loaded with the address of a default interrupt routine containing only RESET instruction. If the interrupt vector table is loaded with an erroneous address or the address that does not mean any routine, the erroneous address trap will be generated which may lead to RESET.

The trap conditions can only be detected when the trap occurs. The routing generated by the trap should be able to remove the error that lead to it. Each trap source has a fixed priority, ranging from level 8 to level 15. This means that the IPL3 bit is always set during processing of any trap.

The sources of traps are classified in for groups with the interrupt priority growing.

A. Arithmetic error traps are generated if during arithmetic operations the following errors occur:

1. Divide-by-zero is attempted; the operation will be stopped and the divide by zero trap will be generated (interrupt priority level 8),
2. If the arithmetic error trap is enabled, and if in the course of a mathematical operation the overflow of accumulator A or B occurs (carry-out from bit 31) and the protection bits in the accumulators (31-bit saturation mode) are not enabled (interrupt priority level 9),
3. If the arithmetic error trap is enabled, and if in the course of a mathematical operation a catastrophic overflow of the accumulator A or B occurs (carry-out from bit 39); the accumulator saturation is prevented (interrupt priority level 10),
4. If druring shift operation the shift value exceeds 16 bits when processing words, or 8 bits when processing bytes (interrupt priority level 11).

B. Address error traps are generated if the following operating situations occur (interrupt priority level 13):

1. A misaligned data word (2 bytes) fetch is attempted (word access with an odd effective address),
2. A data fetch from unimplemented data address space of the microcontroller is attempted,
3. An attempt by the programme to address an unimplemented programme memory location of the microcontroller,
4. Access to an instruction within the vector space is attempted,
5. A branch or jump instruction specifies an address from unimplemented address space of the micronotroller,
6. Upon modification of the programme counter (PC) register, a nonexistent programme memory location, unimplemented in the microcontroller, is addressed.

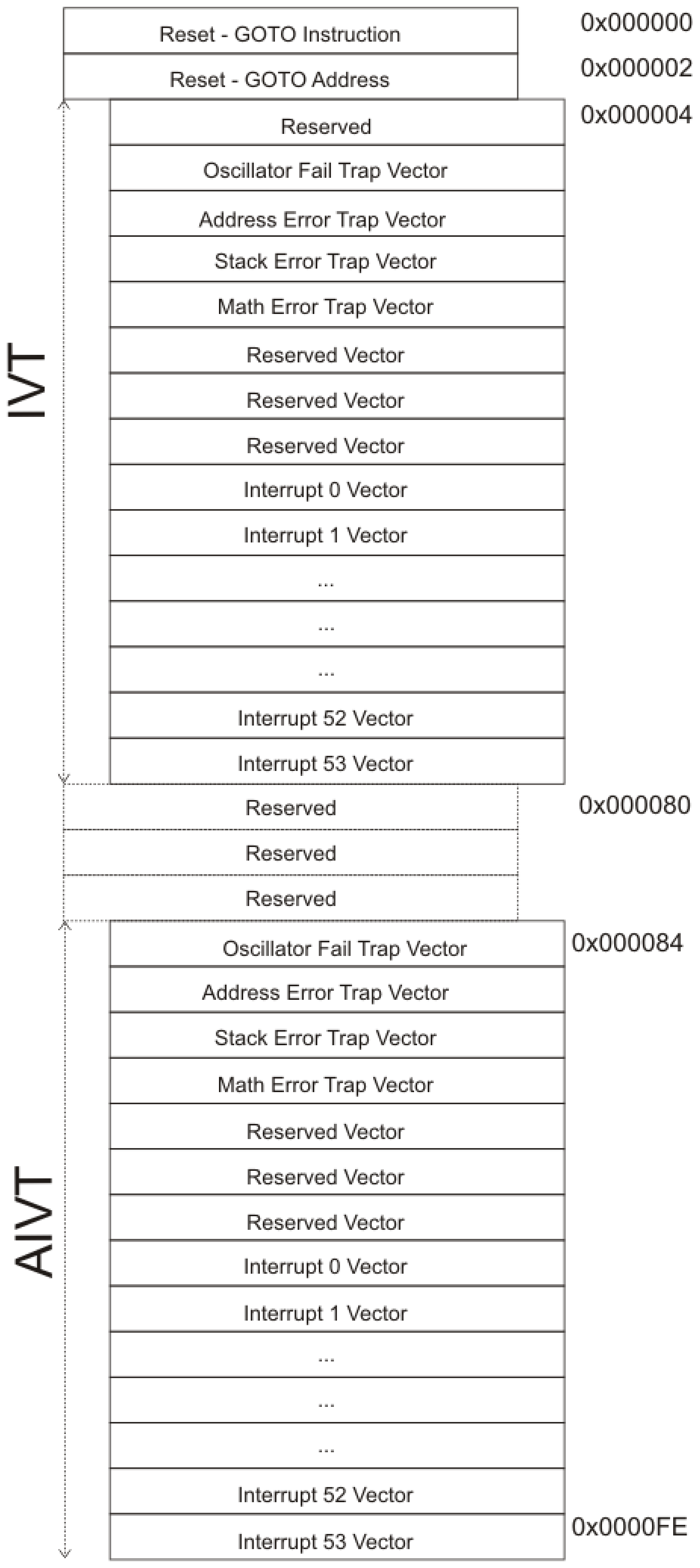
C. Stack error trap is generated if the following events occur (interrupt priority level 13):

1. The stack pointer (SP) value exceeds the value of the SPLIM (Stack Pointer LIMit – set by the user) register, i.e. the stack is exceeded from the upper side,
2. The stack pointer value is below 0x0800, i.e. the stack is exceeded from the lower side.

D. Oscillator failure trap is generated if the external oscillator fails; the reliability of operation of the microcontroller is then based on the internal RC oscillator (interrupt priority level 14).

The traps can be classified in two groups: software and hardware traps. The group of software traps contains traps of priorities from 8 to 11 which arise as a consequence of arithmetic operations. The interrupt routines caused by the software traps are nestable. The group of hardware traps contains traps of priorities from 12 to 15. The interrupt routines caused by the hardware traps are not nestable. The highest priority trap is generated when two hardware traps are in conflict.

The organization of the interrupt vector table and alternate vector table in the programme memory of the family of dsPIC30F microcontrollers is shown in Fig. 3-1.



Reset - GOTO Instruction
0x000000
Reset - GOTO Address
0x000002
Reserved
0x000004
Oscillator Fail Trap Vector
Address Error Trap Vector
Stack Error Trap Vector
Math Error Trap Vector
Reserved Vector
Reserved Vector
Reserved Vector
Interrupt 0 Vector
Interrupt 1 Vector
...
...
...
Interrupt 52 Vector
Interrupt 53 Vector
IVT
Reserved
0x000080
Reserved
Reserved
Oscillator Fail Trap Vector
0x000084
Address Error Trap Vector
Stack Error Trap Vector
Math Error Trap Vector
Reserved Vector
Reserved Vector
Reserved Vector
Interrupt 0 Vector
Interrupt 1 Vector
...
...
...
Interrupt 52 Vector
Interrupt 53 Vector
0x0000FE
AIVT

Fig. 3-1 Organization of the interrupt vector table and alternate interrupt vector table in the program memory of the microcontroller family dsPIC30

For handling interrupts it is necessary to know the interrupt sequence. The interrupt sequence begins by writing an interrupt request, from peripherals or external signals, in the interrupt request bit of the IFSx registers at the beginning of each instruction cycle. An interrupt request (IRQ) is identified through setting the corresponding bit in the IFSx register. The interrupt is generated only if it is enabled, i.e. if the corresponding bit in the IECx register is set. In the course of executing the current instruction cycle the priority levels of all pending interrupts are evaluated. If the priority level of a pending interrupt is higher than the priority level currenty processed by the microcontroller (defined by the bits IPL<3:0>), an interrupt is generated.

Upon generating an interrupt, the microcontroller saves on the software stack the current PC value and the low byte of the processor status register (SRL). The byte SRL contains the priority level of the interrupt being processed until then. After that the microcontroller sets the new interrupt priority level in the STATUS register, equal to the level of the generated interrupt. In this way generating any interrupts of a lower level is prevented. The microcontroller does not automatically take care of the rest of the processor context. This task is undertaken by the compiler who saves PC, SRL, and the values of other registers significant for further execution of the programme.

The instruction **RETFIE** (**RET**urn **F**rom **I**nt**E**rrupt) denotes return from interrupt and unstacks the PC return address and the low byte of the processor status register in order to return the microcontroller to the state and priority level existing prior to the interrupt sequence.

Attention!
The user could lower the priority level of the interrupt being processed by writing a new value in the STATUS register (SR) while processing the interrupt, but the interrupt routine has to clear the interrupt request bit in the IFSx register in order to prevent a recurrent generation of interrupts. The recurrent generation of interrupts could easily lead to stack overflow and the generation of a stack error trap and the microcontroller reset.

NOTE: While processing interrupts the bit IPL3 is always zero. It is set only while processing the processor traps.

For handling interrups in practice, it is necessary to know that the family of microcontrollers dsPIC30F supports up to 5 external interrupt sources, INT0 – INT4. These inputs are edge sensitive, i.e they require transition from low to high logic level or vice versa in order to generate an interrupt. The INTCON2 register contains bits INT0EP – INT4EP defining the polarity of the sensitive edge, i.e. rising or falling edge of the signal.

Very often interrupts are used for waking-up the microcontroller from SLEEP or IDLE mode if the microcontroller is in these states when an interrupt is generated. The SLEEP or IDLE states are used when the minimization of the power consumption or minimization of the noise in the course of AD converter measurement are required. The microcontroller will wake-up and initiate the execution of interrupt routine only if the interrupt request is enabled in the IECx register.

The following example shows how a trap operates in practice

Example:

A programme contains a mistake. There is a chance of divide by zero. Since this is not correct, it is possible that this mistake, if there was not the function ment to react, wouls reset dsPIC. This example uses the code from the example of interrupts. On each increasing edge of the bit RF6 the value at port B is decremented by 1 and thus approaches zero which is undesirable because of the danger of dividing by the value of port B in the main programme. The programme is as follows:

```
int a;



void IntDet() org 0x0014{    //vector INT0

  LATB--;                    //portB is decremented

  IFS0.F0 = 0;               //interrupt flag cleared

}



void TrapTrap() org 0x000C{

  INTCON1.F4 = 0;

  /*the problem is solved by setting

    port B to a nonzero value*/

  LATB = 3;

  LATD++;

}



void main(){

  TRISB = 0;

  TRISD = 0;
  TRISF = 0xFFFF;

  LATB = 3;

  LATD = 0;

  IFS0 = 0;     //interrupt flag cleared

  INTCON1 = 0;  //trap flag cleared

  IEC0 = 1;     //interrupt on rising edge INT0 (RF6) enabled

  while(1){

     a = 256 / LATB; //if LATB=0 error occured and TrapTrap is called

  }

}
```

A description of the interrupt registers of the microcontroller dsPIC30F4013 is presented at the end of this chapter.

NOTE: Reading bits with no function assigned gives ‘0’.

Name	**ADR**	**15**	**14-11**	**10**	**9**	**8**
INTCON1	0X0080	NSTDIS	-	OVATEN	OVBTEN	COVTE

Table 3-2. Interrupt control register - INTCON1

7-5	**4**	**3**	**2**	**1**	**0**	**Reset State**
-	MATHERR	ADDRER	STKERR	OSCFAIL	-	0x0000

Table 3-2. continued

```
NSTDIS – Interrupt nesting disable bit
OVATEN – Accumulator A overflow trap enable bit
OVBTEN – Accumulator B overflow trap enable bit
COVTE – Catastrophic overflow trap enable bit
MATHERR – Arithmetic error trap status bit
ADDRERR – Address error trap status bit
STKERR – Stack error trap status bit
OSCFAIL – Oscillator failure trap status bit
```

name	**ADR**	**15**	**14**	**13**	**12**	**11**	**10**	**9**
INTCON2	0x0082	ALTIVT	-	-	-	-	-	-

Table 3-3. Interrupt control register 2 - INTCON2

8	7	6	5	4	3	2	1	0	Reset State
-	-	-	-	-	-	INT2EP	INT1EP	INT0EP	0x0000

Table 3-3. continued

```
ALTIVIT – Enable alternate interrupt vector table bit
INT0EP, INT1EP, INT2EP – External interrupt edge detect polarity bits
```

name	ADR	15	14	13	12	11	10	9	8
IFS0	0x0084	CNIF	MI2CIF	SI2CIF	NVMIF	ADIF	U1TXIF	U1RXIF	SPI1IF

Table 3-4. Interrupt flag status register - IFS0

7	6	5	4	3	2	1	0	Reset State
T3IF	T2IF	OC2IF	IC2IF	T1F	OC1IF	IC1IF	INT0IF	0x0000

Table 3-4. continued

```
CNIF – Input change notification flag status bit
MI2CIF – I2C module (master mode) transmitter interrupt flag status bit
SI2CIF – I2C module (slave mode) receiver interrupt flag status bit
NVMIF – Non-volatile memory write complete interrupt flag status bit
ADIF – A/D conversion complete interrupt flag status bit
U1TXIF – UART1 transmitter interrupt flag status bit
U1RXIF – UART1 receiver interrupt flag status bit
SPI1IF – SPI1 interrupt flag status bit
T3IF – Timer 3 interrupt flag status bit
T2IF - Timer 2 interrupt flag status bit
OC2IF – Output compare channel 2 interrupt flag status bit
IC2IF – Input capture channel 2 interrupt flag status bit
T1IF – Timer1 interrupt flag status bit
OC1IF – Output compare channel 1 interrupt flag status bit
IC1IF – Input captue channel 1 interrupt flag status bit
INT0IF – External interrupt 0 flag status bit
```

name	ADDR	15-12	11	10	9	8	7
IFS1	0x0086	-	C1IF	-	U2TXIF	U2RXIF	INT2IF

Table 3-5. Interrupt flag status register - IFS1

6	5	4	3	2	1	0	Reset State
T5IF	T4IF	OC4IF	OC3IF	IC8IF	IC7IF	INT1IF	0x0000

Table 3-5. continued

```
C1IF – CAN1 (combined) interrupt flag status bit
U2TXIF – UART2 transmitter interrupt status bit
U2RXIF – UART2 receiver interrupt flag status bit
INT2IF – External interrupt 2 flag status bit
T5IF – Timer5 interrupt flag status bit
T4IF – Timer4 interrupt flag status bit
OC4IF – Output compare channel 4 interrupt flag status bit
OC3IF- Output compare channel 3 interrupt flag status bit
IC8IF – Input capture channel 8 interrupt flag status bit
IC7IF - Input capture channel 7 interrupt flag status bit
INT1IF – External interrupt 1 flag status bit
```

name	ADDR	15	14	13	12	11	10	9
IFS2	0x0088	-	-	-	-	-	LVDIF	DCIIF

Table 3-6. Interrupt flag status register 2 - IFS2

8	7	6	5	4	3	2	1	0	Reset State
-	-	-	-	-	-	-	-	-	0x0000

Table 3-6. continued

LVDIF – Programmable low voltage detect interrupt flag status bit
DCIIF – Data converter interface interrupt flag status bit

name	ADDR	15	14	13	12	11	10	9	8
IEC0	0x008C	CNIE	MI2CIE	SI2CIE	NVMIE	ADIE	U1TXIE	U1RXIE	SPI1IE

Table 3-7. Interrupt enable control register 0 - IEC0

7	6	5	4	3	2	1	0	Reset State
T3IE	T2IE	OC2IE	IC2IE	T1E	OC1IE	IC1IE	INT0IE	0x0000

Table 3-7. continued

CNIE – Input change notification interrupt enable bit
MI2CIE – I2C module (master mode) transmitter interrupt enable bit
SI2CIE – I2C I2C module (slave mode) receiver interrupt enable bit
NVMIE – Non-volatile memory write complete interrupt enable bit
ADIF – A/D conversion complete interrupt enable bit
U1TXIE – UART 1 transmitter interrupt enable bit
U1RXIE – UART 1 receiver interrupt enable bit
SPI1IE – SPI1 interrupt enable bit
T3IE – Timer3 interrupt enable bit
T2IE – Timer2 interrupt enable bit
OC2IE – Output compare channel 2 interrupt enable bit
IC2IE – Input capture channel 2 interrupt enable bit
T1IE – Timer1 interrupt enable bit
OC1IE – Output compare channel 1 interrupt enable bit
IC1IE – Input capture channel 1 interrupt enable bit
INT0IE – External interrupt 0 enable bit

name	ADDR	15-12	11	10	9	8	7
IEC1	0x008E	-	C1IE	-	U2TXIE	U2RXIE	INT2IE

Table 3-8. Interrupt enable control register 1 - IEC1

6	5	4	3	2	1	0	Reset State
T5IE	T4IE	OC4IE	OC3IE	IC8IE	IC7IE	INT1IE	0x0000

Table 3-8. continued

C1IE – CAN1 (combined) interrupt enable bit
U2TXIE – UART2 transmitter interrupt enable bit
U2RXIE – UART2 receiver interrupt enable bit
INT2IE – External interrupt 2 enable bit
T5IE – Timer5 interrupt enable bit

T4IE – Timer4 interrupt enable bit
OC4IE – Output compare channel 4 interrupt enable bit
OC3IE – Output compare channel 3 interrupt enable bit
IC8IE – Input capture channel 8 interrupt enable bit
IC7IE – Input capture channel 7 interrupt enable bit
INT1IE – External interrupt 1 enable bit

name	ADDR	15	14	13	12	11	10	9	8	7
IEC2	0x0090	-	-	-	-	-	-	-	LVDIE	DCIIE

Table 3-9. Interrupt enable control register 2 - IEC2

8	7	6	5	4	3	2	1	0	Reset State
-	-	-	-	-	-	-	-	-	0x0000

Table 3-9. continued

LVDIE – Programmable low voltage detect interrupt enable bit
DCIIE – Data converter interface interrupt enable bit

name	ADDR	15	14	13	12	11	10	9	8
ICP0	0x0094	-	T1IP<2:0>			-	OC1IP<2:0>		
ICP1	0x0096	-	T3IP<2:0>			-	T2IP<2:0>		
ICP2	0x0098	-	ADIP<2:0>			-	U1TXIP<2:0>		
ICP4	0x009C	-	OC3IP<2:0>			-	IC8IP<2:0>		
ICP5	0x009E	-	INT2IP<2:0>			-	T5IP<2:0>		
ICP6	0x00A0	-	C1IP<2:0>			-	SPI2IP<2:0>		
ICP7	0x00A2	-	-	-	-	-	-	-	-
ICP8	0x00A4	-	-	-	-	-	-	-	-
ICP9	0x00A6	-	-	-	-	-	-	-	-
ICP10	0x00A8	-	-	-	-	-	LVDIP<2:0>		

Table 3-10. Special function registers associated wuth interrupt controllers

7	6	5	4	3	2	1	0	Reset State
-	IC1IP<2:0>			-	INT0IP<2:0>			0x0000
-	T2IP<2:0>			-	IC2IP<2:0>			0x0000
-	U1TXIP<2:0>			-	SPI1IP<2:0>			0x0000
-	SI2CIP<2:0>			-	NVMIP<2:0>			0x0000
-	IC7IP<2:0>			-	INT1IP<2:0>			0x0000
-	T4IP<2:0>			-	OC4IP<2:0>			0x0000
-	U2TXIP<2:0>			-	U2RXIP<2:0>			0x0000
-	-	-	-	-	-	-	-	0x0000
-	-	-	-	-	-	-	-	0x0000
-	-	-	-	-	-	-	-	0x0000
-	DCIIP<2:0>			-	-	-	-	0x0000

Table 3-10. continued

T1IP<2:0> - Timer1 interrupt priority bits
OC1IP<2:0> - Output compare channel 1 interrupt priority bits
IC1IP<2:0> – Input capture channel 1 interrupt priority bits
INT0IP<2:0> - External interrupt 0 priority bits
T3IP<2:0> - Timer3 interrupt priority bits
T2IP<2:0> - Timer2 interrupt priority bits
OC2IP<2:0> - Output compare channel 2 interrupt priority bits
IC2IP<2:0> -Input capture channel 2 interrupt priority bits
ADIP<2:0> - A/D conversion complete interrupt priority bits
U1TXIP<2:0> - UART1 transmitter interrupt priority bits
U1RXIP<2:0> - UART1 receiver interrupt priority bits
SPI1IP<2:0> - SPI1 interrupt priority bits
CNIIP<2:0> - Input change notification interrupt priority bits
MI2CIP<2:0> - I2C module (master mode) transmitter interrupt priority level bits
SI2CIP<2:0> - I2C module (slave mode) receiver interrupt priority level bits
NVMIP<2:0> - Non-volatile memory write interrupt priority bits
OC3IP<2:0> - Output compare channel 3 interrupt priority bits
IC8IP<2:0> - Input compsre channel 8 interrupt priority bits
IC7IP<2:0> - Input compare channel 7 interrupt priority bits
INT1IP<2:0> - External interrupt 1 priority bits
INT2IP<2:0> - External interrupt 2 priority bits
T5IP<2:0> - Timer5 interrupt priority bits
T4IP<2:0> - Timer4 interrupt priority bits
OC4IP<2:0> - Output compare channel 4 interrupt priority bits
C1IP<2:0> - CAN1 (combined)interrupt priority bits
SPI2IP<2:0> - SPI2 interrupt priority bits
U2TXIP<2:0> - UART2 transmitter interrupt priority bits
U2RXIP<2:0> - UART2 receiver interrupt priority bits
LVDIP<2:0> - Programmable low voltage detect interrupt priority bits
DCIIP<2:0> - Data converter interface interrupt priority bits

Chapter4: Timers

Introduction

Timers are basic peripherals of each microcontroller. Depending upon the model, the dsPIC30F family offers several 16-bit timer modules. Microcontroller dsPIC30F4013 contains 5 timer modules.

Each timer module contains one 16-bit timer/counter consisting of the following registers:

- TMRx – 16-bit timer counter register,
- PRx – 16-bit period register containing value of the period,
- TxCON – 16-bit control register for selecting mode of the timer.

Each timer module also has the associated bits for interrupt control:

- TxIE – interrupt enable control bit,
- TxIF – interrupt flag status bit,
- TxIP<2:0> - three interrupt priority control bits (in the interrupt register IPCx).

Most of the timers in the family of dsPIC30F microcontrollers have the same functional circuitry. Depending of their functional differences, they are classified into three types: A, B, or C. Timers of the types B and C can be combined to form a 32-bit timer.

4.1 Type A timer

Type A timer is available on most dsPIC30F devices. Timer1 is a type A timer. A type A timer has the following unique features over other types:

- can be operated from the device low power 32 kHz oscillator,
- can be operated in an asynchronous mode from an external clock source.

The unique feature of a type A timer is that it can be used for real-time clock (RTC) applications. A block diagram of the type A timer is shown in Fig. 4-1.

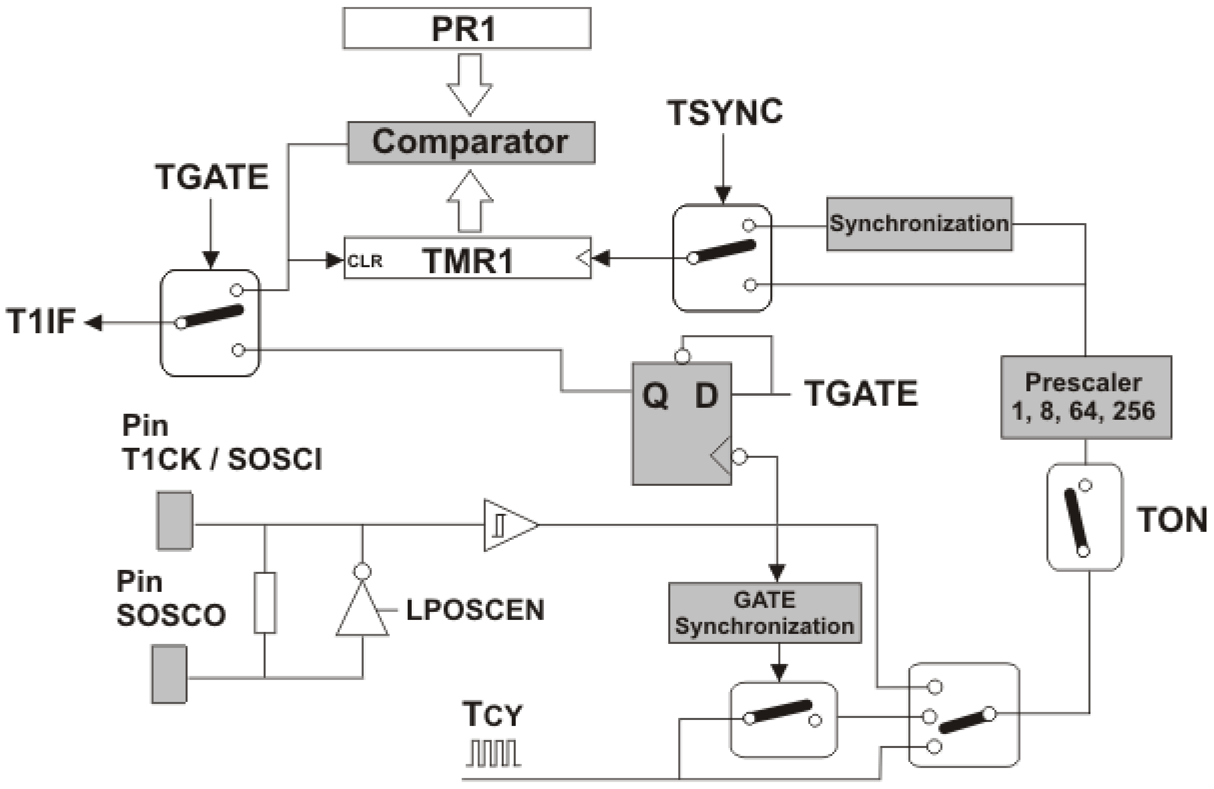


Fig. 4-1 Type A timer block diagram

Type A timer is a general purpose (GP) 16-bit timer and in the microcontroller dsPIC30F4013 it is denoted as timer1 module.

Timer1 module is a 16-bit timer primarily intended for use as a real time counter, or universal period meter, or event counter in the free-running mode. Timer1 can be configured to operate as:

1. a 16-bit timer,
2. a 16-bit synchronous counter,
3. a 16-bit asynchronous counter.

Also, timer1 has the ability to operate as a gated timer, select a prescaler, operate during SLEEP or IDLE states of the microcontroller, as well as generate an interrupt request when the value of the register TMR1 equals that of the period register PR1. The condition for generatig an interrupt request could be the falling edge of an external gating signal (TGATE).

Control of the operation of the timer1 module is determined by setting bits in the 16-bit configuration special function register (SFR) T1CON.

4.1.1 16-bit timer mode

In the 16-bit timer mode the value of the TMR1 counter is incremented by each instruction cycle until it is equal to the preset value of the PR1 register, when the counter is reset to '0' and restarted. At this moment an interrupt request for timer1 module T1IF (in the register IFS0) is generated. Processing of this request depends on the interrupt T1IE enable bit (in the IEC0 register). Timer module continues to operate during the interrupt routine.

Timer1 module can operate during the IDLE state if the TSIDL bit (T1CON<13>) is reset, but if this bit is set, the counter continues to operate after the processor is waken-up from IDLE state.

Attention!
Interrupt of timer1 module is generated upon an interrupt request set by timer1 module only if the interrupt is enabled by setting the T1IE bit in the IEC0 register. The interrupt request bit T1IF in the IFS0 register has to be reset by the software after the interrupt is generated.

The following example demonstrates how timer1 module can be used in the 16-bit timer mode.

Example:

Switch on and off a LED diode at port D approximately 4 times per second. This example uses timer1 module having clock 256 times slower then the dsPIC clock. At each 10 000 clocks timer1 module calls interrupt routine Timer1Int and changes the value at port D.

```
void Timer1Int() org 0x1A{// Timer1 address in the interrupt vector table

  LATD = ~PORTD;          // PORTD inversion

  IFS0 = IFS0 & 0xFFF7;   // Interrupt flag reset

}
```

```
void main(){

  TRISD = 0;              // PORTD is output

  LATD  = 0xAAAA;         // Set initial value at port D

  IPC0  = IPC0 | 0x1000;  // Priority level is 1

  IEC0  = IEC0 | 0x0008;  // Timer1 interrupt enabled

  PR1   = 10000;          // Interrupt period is 10000 clocks

  T1CON = 0x8030;         // Timer1 enabled (internal clock divided by 256)

  while(1) asm nop;       // Endless loop

}
```

How does one calculate the period of interrupt requests? Let the internal clock be set to 10MHz. The period is 100ns. Since the clock is divided by 256 (prescaler reduces the clock 1:256) to form the timer clock, it follws that the the timer clock is 100*256=25600ns i.e. 25.6µs. At each 10000 clocks an interrupt is requestd, i.e. at each 256ms or approximately 4 times per second.

T = 10000*25.6µs = 256ms ~ ¼s.

4.1.2 16-bit synchronous counter mode

In the 16-bit synchronous counter mode the counter value TMR1 increments on every rising edge of an external clock source. In this mode the phase of the internal clock is synchronized with the external clock. When the value of the TMR1 counter is equal to the preset value of the PR1 register, the counter TMR1 is reset and starts counting from ‘0’. At this moment an interrupt request for timer1 module T1IF (in the register IFS0) is generated. Processing of this request depends on the interrupt T1IE enable bit (in the IEC0 register). Timer module continues to operate during the interrupt routine.

The purpose of the timer1 module is to allow measurement of the periods of very fast clock signals, e.g. autodetection of the speed of communication of the universal serial inteface UART.

The timer1 module could operate in this mode during IDLE state if bit TSIDL (T1CON<13>) is reset; if this bit is set, the counter continues operation after the processor is waken-up from IDLE state.

The following example demonstrates how timer1 module can be used in the synchronous counter mode.

Example:

Count every tenth pulse and increment the value at port D. In this example timer1 is used for counting external clock pulses at pin T1CK. After ten pulses interrupt Timer1Int occurs and the vaule at port D is incremented. Block diagram of Fig. 4-2 shows the interconnection between the timer1 and external clock source.

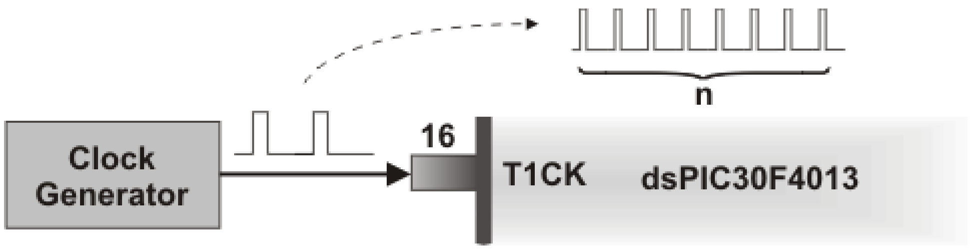


Fig. 4-2 Block diagram of the connection of an external clock source to dsPIC30F4013

```
void Timer1Int() org 0x1A{  // Timer1 address in the interrupt vector table

  LATD  = ~PORTD;           // Value at PORTD is incermented

  IFS0  = IFS0 & 0xFFF7;    // Interrupt request is reset

}



void main(){

  TRISD  = 0;               // PORTD is output

  TRISC  = 0x4000;          // PORT<14>=1 T1CK input pin

  LATD = 0;                 // Set initial value at port D

  IPC0 = IPC0 | 0x1000;     // Priority level is 1

  IEC0 = IEC0 | 0x0008;     // Timer1 interrupt enabled

  PR1  = 10;                // Interrupt period is 10 clocks

  T1CON  = 0x8006;          // Timer1 is synchronous counter of external
pulses

    while(1) asm nop;       // Endless loop
}
```

How does one configure timer1 module to operate in the 16-bit synchronous mode?
Prescaler ratio 1:1 is selected, external clock TCS=1 is enabled, and operation of timer1 module TON=1 is enabled (T1CON = 0x8006;). By setting bit TRISC<14>=1 the pin PORTC<14>=1 is configured as input.

4.1.3 16-bit asynchronous counter mode

In the 16-bit asynchronous counter mode the counter value TMR1 increments on every rising edge of an external clock source, but the phase of the internal clock is not synchronized with the external clock. When the value of the TMR1 counter is equal to the preset value of the PR1 register, the counter TMR1 is reset and starts counting from '0'. At this moment an interrupt request for timer1 module T1IF (in the register IFS0) is generated. Processing of this request depends on the interrupt T1IE enable bit (in the IEC0 register). Timer module continues to operate during the interrupt routine.

The timer1 module could operate in this mode during IDLE state if bit TSIDL (T1CON<13>) is reset; if this bit is set, the counter continues operation after the processor is waken-up from IDLE state.

Example:

Count each 800th pulse and increment the value at port D. In this example timer1 is used to count each 8th pulse of an external clock at the external pin T1CK. After 100 pulses the interrupt routine Timer1Int is called and the value at port D is incremented. Block diagram of Fig. 4-3 shows the interconnection between the timer1 and external clock source.

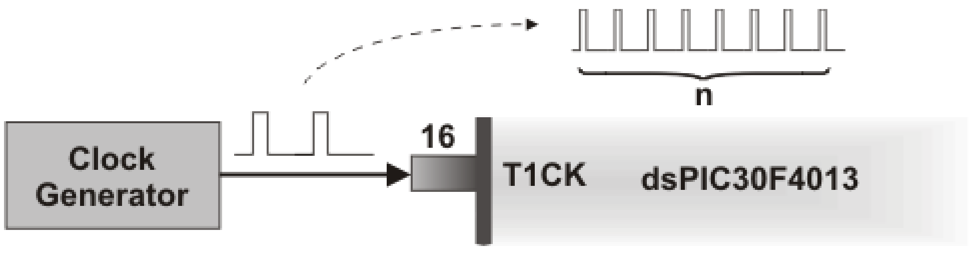


Fig. 4-3 Block diagram of the connection of an external clock source to dsPIC30F4013

```
void Timer1Int() org 0x1A{ // Timer1 address in the interrupt vector table

  LATD++;                  // Value at PORTD is incremented

  IFS0 = IFS0 & 0xFFF7;    // Interrupt request is reset

}
```

```
void main(){

  TRISD = 0;               // PORTD is output

  TRISC = 0x4000;          // PORT<14>=1 T1CK is input pin

  LATD  = 0;               // Set initial value at port D

  IPC0  = IPC0 | 0x1000;   // Priority level is 1

  IEC0  = IEC0 | 0x0008;   // Timer1 interrupt enabled

  PR1   = 100;             // Interrupt period is 100 clocks

  T1CON = 0x8012;          // Timer1 is synchronous counter of external 
pulses

  while(1) asm nop;        // Endless loop

}
```

4.1.4 Gated timer mode

When timer1 module operates in the 16-bit timer mode, it can be configured to allow the measurement of duration of an external gate signal (gate time accumulation). In this mode the counter TMR1 is incremented by internal instruction clock (TCY) as long as the external GATE signal (pin T1CK) is at the high logical level. In order that timer1 module operates in this mode it is required that the control bit TGATE (T1CON<6>) is set, internal clock (TCS=0) is selcted, and the operation of the timer is enabled (TON=1).

The timer1 module could operate in this mode during IDLE state if bit TSIDL (T1CON<13>) is reset; if this bit is set, the counter continues operation after the processor is waken-up from IDLE state.

Example:

Use timer1 in the gated time accumulation mode.The enable GATE signal is applied to the pin T1CK. Measure the width of the signal and display the result at port D. Block diagram of interconnection of timer1 and an external clock source is shown in Fig. 4-4.

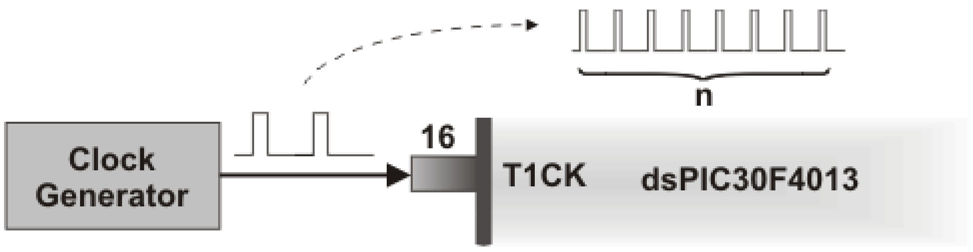


Fig. 4-4 Block diagram of the connection of an external clock source to dsPIC30F4013

```
void Timer1Int() org 0x1A{// Timer1 address in the interrupt vector table

  LATD = TMR1;            // Pulse duration is displayed at port D

  IFS0 = IFS0 & 0xFFF7;   // Interrupt request is processed

}



main(){

  TRISD = 0;              // PORTD is output

  TRISC = 0x4000;         // PORT<14>=1 T1CK is input pin

  LATD  = 0;              // Set initial value at port D

  IPC0  = IPC0 | 0x1000;  // Priority level is 1

  IEC0  = IEC0 | 0x0008;  // Timer1 interrupt enabled

  PR1   = 0xFFFF;         // Period is maximum

  T1CON = 0x8040;         // Timer1 is enabled, internal clock until T1CK=1

  while (1) asm nop;      // Endless loop

}
```

Why the register PR1 is set to the maximum value? The main reason is to allow measurement of as long as possible time intervals, i.e. the interrupt does not come as a consequence of equalisation of the TMR1 and PR1 registers but, if possible, as a consequence of the falling edge of the GATE signal.

In the operation of the timer1 module the applications often require the use of the prescaler which allows that the clock can be reduced by the ratio 1:1, 1:8; 1:64, or 1:256. In this way the scope of applications of the timer1 module is widened considerably. The prescaler selection is done by setting the control bits TCKPS<1:> (T1CON<5:4>). The prescaler counter is reset every time of writing in the counter register TMR1, or control register T1CON, or after microcontroller reset. However, it is important to note that the prescaler counter can not be reset when the timer1 module is interrupted (TON=0) since the prescaler clock is stoped.

Attention!
The counter register TMR1 is not reset when a new value is written in the control regster T1CON. The contents of the TMR1 register has to change after writing a new value in this register.

The operation of the timer1 module in SLEEP mode is possible only if the following conditions are fullfiled:

- the operation of the timer1 module is enabled (TON=1),
- timer1 module uses an external clock,
- control bit TSYNCH (T1CON<2>) is reset defining an asynchronous external clock source. i.e. the clock is independent of the internal microcontroller clock.

In this way it is allowed that the timer1 module continues counting as long as the register TMR1 is equal to the preset (period) register PR1. Then, TMR1 is reset and an interrupt is generated. An interrupt request for the timer1 module, if the enable bit is set, can wake-up the microcontroller from SLEEP mode.

The timer1 module generates an interrupt request when the values of the counter register TMR1 and preset (period) register PR1 are equal. The interrupt request is effected by setting bit T1IF in the interrupt register IFS0. If the timer1 module interrupt enable bit T1IE in the interupt register IEC0 is set, an interrupt is generated. **The bit T1IF has to be software reset in the interrupt routine.**

If the timer1 module operates in the gated time accumulation mode, then the interrupt request will be generated at each falling edge of the external GATE signal, i.e. at the end of each accumulation cycle.

4.1.5 Real-Time Clock (RTC) operation mode

The timer1 module can be adjusted to operate in the Real-Time Clock operation mode, i.e. as a real time clock. In this way one obtains the information of the time instants (hours, minutes, seconds) serving for the evidence of events. The main characteristics of the RTC mode of operation of the timer1 module are: use of the 32kHz oscillator, 8-bit prescaler, low power, and the ability of generating an RTC interrupt request. Like all timer1 module operation modes, this mode is set by the control bits in the register T1CON. While the timer1 module uses the clock of the 32kHz oscillator for operation in the RTC mode, the remaining of the microcontroller has the ability to operate with another clock which is adjustable by the control bits in the control register FOSC.

In this mode, when the control bits are TON=1, TCS=1, and TGATE=0, the counter register TMR1 is incremented by each rising edge of the 32kHz low power oscillator until the value of the counter register TMR1 is equal to the preset value of the PR1 register; the counter register TMR1 is then reset t? '0'.

Attention!
In order that timer1 module operates correctly in the RTC mode, the control bit TSYNC has to be reset. By setting TSYNC=0 the asynchronous mode of the timer1 module is set. Also, the bit LPOSCEN (T1CON<1>) has to be set to disable all other operation modes except RTC and enable wake-up of the microcntroller from SLEEP state by the timer1 module. It is very important that the TSIDL bit is reset to allow operation of timer1 module during IDLE state.

With this configured timer1 module, in SLEEP state the TRC will continue operation clocked from the 32kHz oscillator and the control bits will not be changed.

When the condition for generation of an interrupt request is fulfilled (TMR1=PR1), the bit T1IF in the interrupt register IFS0 is set. If the corresponding interrupt is enabled (the enable bit T1IE in the IEC0 register is set), an interrupt of the microcontroller is generated. **During the interrupt routine the bit T1IF has to be reset, otherwise no other interrupt request could be detected by timer1 module.**

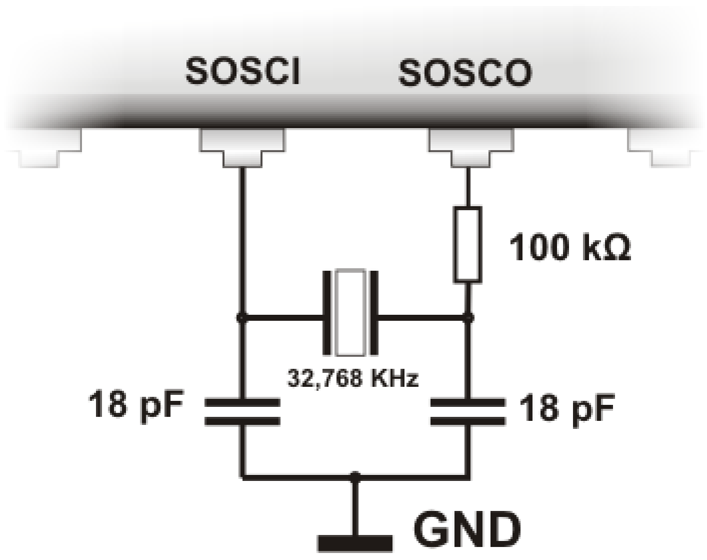


Fig. 4-5 Connection of a crystal oscillator in the RTC mode

name	**ADR**	**15**	**14**	**13**	**12-7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Reset State**
TMR1	0x0100	Timer 1 Register											0x0000
PR1	0x0102	Period Register 1											0xFFFF
T1CON	0x0104	TON	-	TSIDL	-	TGATE	TCKPS1	TCKPS0	-	TSYNC	TCS	-	0x0000

Table 4-1 Registers of timer1 module

```
TON – Timer1 module on bit (TON=1 starts timer, TON=0 stops timer)
TSIDL – Stop in IDLE mode bit (TSIDL=1discontinue module operation when
        microcontroller enters IDLE mode,
TSIDL = 0 continue module operation in IDLE mode)
TGATE – Timer gated time accumulation mode enable bit (TCS must be set to
logic 0 when TGATE=1)
TCKPS<1:0> - Timer input clock prescale select bits
        00 – 1:1 prescale value
        01 – 1:8 prescale value
        10 – 1:64 prescale value
        11 – 1:256 prescale value
TSYNC – Timer external clock input synchronization select bit
(TSYNC=1 synchronize external clock input, TSYNC=0 do not synchronize
external clock input)
TCS – Timer clock source select bit
(TCS=1 external clock from pin T1CK, TCS=0 internal clock Fosc/4)
```

NOTE: Unimplemented bits read as ‘0’.

4.2 Type B timer

Type B timer is a 16-bit timer present in most devices of the dsPIC30F family of microcontrollers. It is denoted as the timer2 module and timer4 module like in dsPIC30F4013 microcontroller. Type B timer has the following specific features:

- a type B timer can be concatenated with a type C timer to form a 32-bit timer,
- the clock synchronization for a type B timer is performed after the prescaler.

Type B timer functional block diagram is shown in Fig. 4-6.

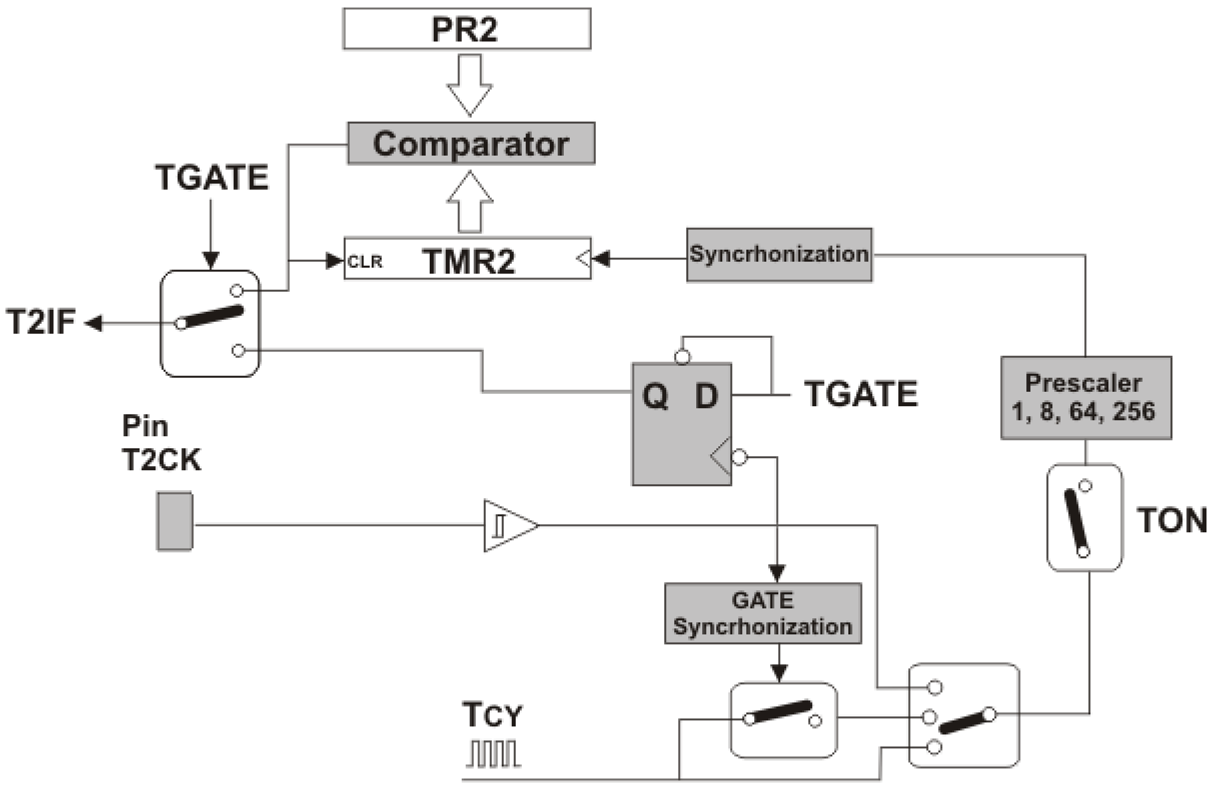


Fig. 4-6 Type B timer functional block diagram (timer2 module)

4.3 Type C timer

Type C timer is a 16-bit timer. Most often it is used together with type B timer to form a 32-bit general purpose timer. Timers of this type, if present, in the dsPIC30F family of microcontrollers are denoted as the timer3 module and timer5 module like in dsPIC30F4013 microcontroller. Type C timer has the following specific features:

- a type C timer can be concatenated with a type B timer to form a 32-bit timer,
- at least one type C timer has the ability to trigger an A/D conversion.

Type C timer functional block diagram is shown in Fig. 4-7.

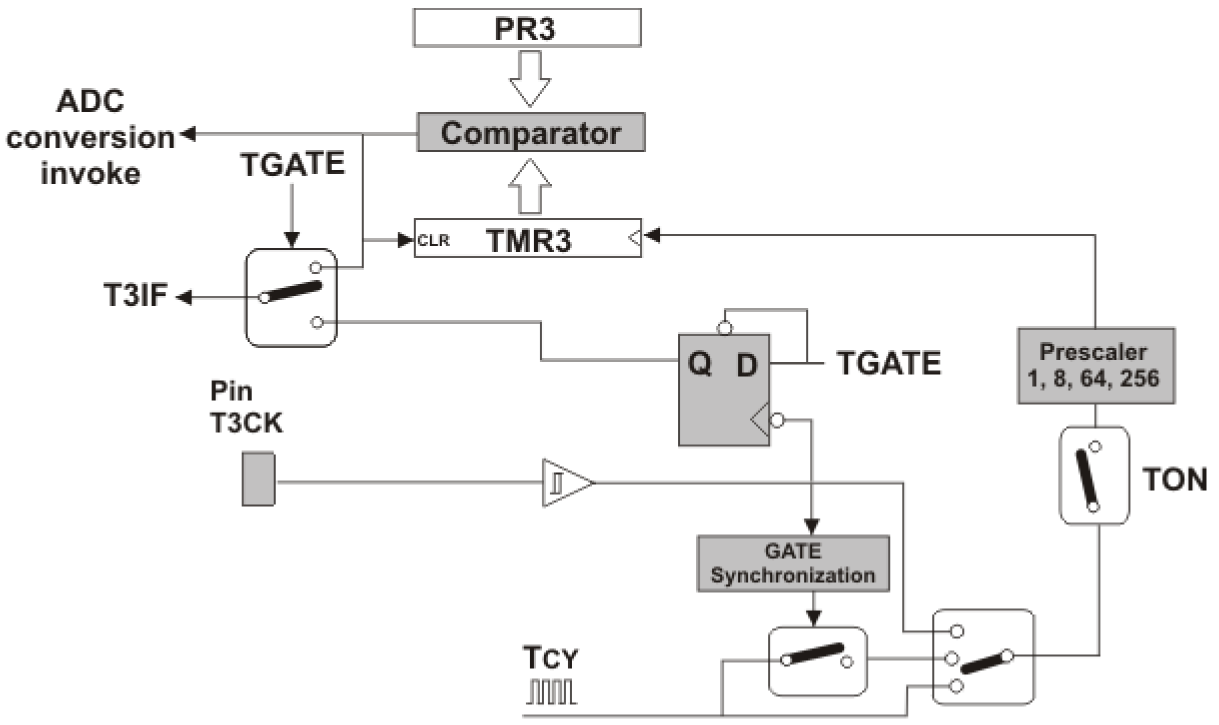


Fig. 4-7 Type C timer functional block diagram (timer 3 module)

Attention!
Pin T3CK does not exist in dsPIC30F4013 devices. The figure shows organization of the timer module of a dsPIC30F6014A device. Also, timer5 module does not have the ability to trigger an A/D conversion.

4.4 Concatenated 32-bit timer

The described type B and C timers, as already mentioned, could be concatenated to form a 32-bit timer. Functional block diagram of the concatenated 32-bit timer is shown in Fig. 4-8.

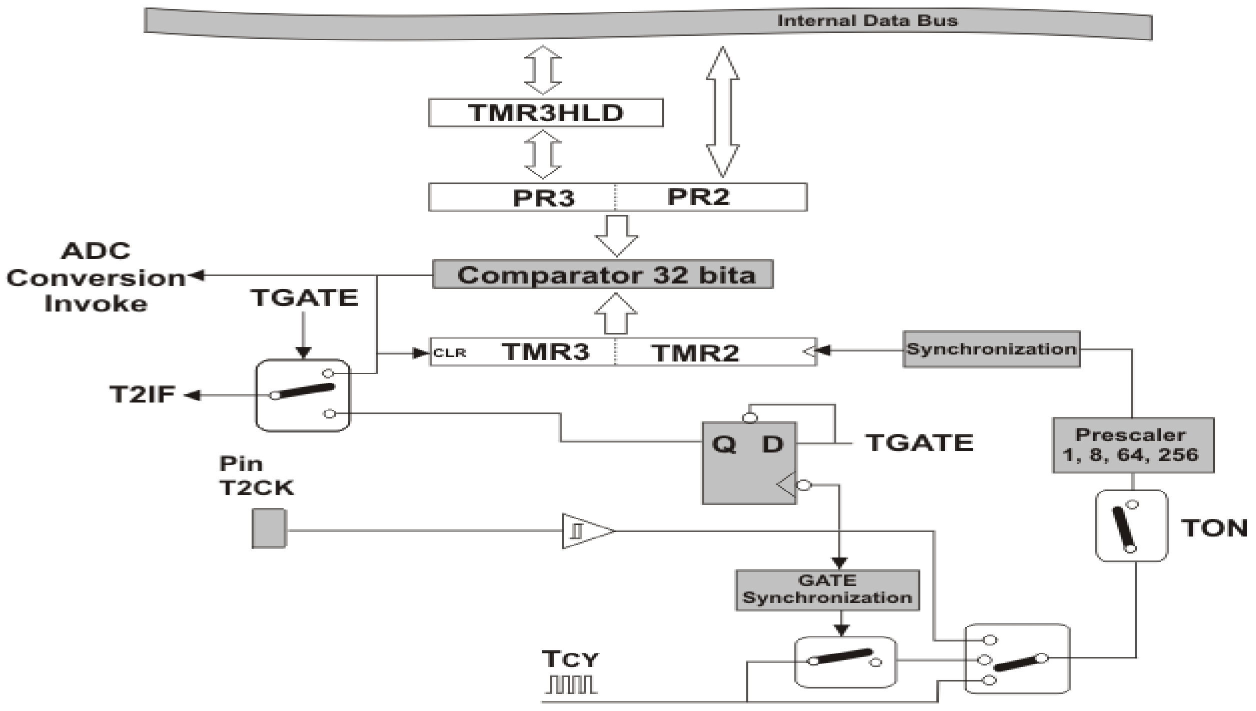


Fig. 4-8 Type C timer functional block diagram (timer 3 module)

The 32-bit timer of Fig. 4-8 is formed by combining the timer2 and timer3 modules. The timer4 and timer5 modules could be combined in a similar way. The formed 32-bit timer is a general purpose (GP) timer and could be configured by the control bits to operate in the following modes:

- 32-bit timer.
- 32-bit synchronous counter.

It is significant to note that the timer2 and timer3 modules could independently operate as 16-bit timers in the all operational modes as the timer1 module, except in the asynchronous counter mode. The same applies for timer4 and timer5 modules.

Timer2 and timer3 modules make use of other peripherals, Input capture or Output compare, in order to realize in a simple way a pulse-width modulated (PWM) signal or an A/D converter for triggering conversion at precise sampling times which is not possible with the timer4 and timer5 modules.

The concatenated 32-bit timer2/3 module has the ability to:

- A/D event trigger,
- operate in gated time accumulation mode,
- prescaler select,
- operate in IDLE state, and
- generate an interrupt request upon the concatenated counter register TMR2/TMR3 is equal with the preset register PR3/PR2.

Adjustment of the mode of operation of the concatenated timer2/3 module is performed by the control bits in T2CON and T3CON.

Attention!
If the timer2 and 3 modules operate independently, the operation modes are adjusted by T2CON and T3CON. If they operate as the concatenated timer2/3 module, then the T2CON control bits are adjusted and T3CION control bits are ignored. The timer2 module clock and gate inputs are used for the concatenated 32-bit timer2/3 module. An interrupt of the 32-bit timer is generated with the T3IF flag and the corresponding T3IE enable bit.

4.4.1 32-bit timer mode

In the 32-bit timer mode the timer is incemented at each instruction cycle until the value of the concatenated counter register TMR3/TMR2 is equal to the concatenated preset register PR3/PR2. Then the counter register is reset to ‘0’ and an interrupt request is generated with the bit T3IF.

NOTE: Synchronous reading of the 32-bit TMR3/TMR2 register is carried out by reading timer2 module 16-bit TMR2 register as the Less Significant word (LS). During reading of TMR2 register the value of the TMR3 register is transferred to the temporary register TMR3HLD. The process of the 32-bit reading is concluded by reading the value of the Most Significant word (MS) from the register TMR3HLD.
The synchronous 32-bit writing is performed in two steps inversly to the reading. At first the higher significant word is written in the TMR3HLD and then the Less significant word is

written in TMR2. During the writing in the TMR2 register the vaules of TMR3HDL and the counter register TMR3 are transferred to TMR2.

The following example demonstrates how the concatenated timer2 and 3 modules can be used in a 32-bit timer.

Example:

Turn on and off a LED diode at port D approximately once every two seconds. The example uses the concatenated timer2 and 3 modules having 256 times slower clock than that of the dsPIC device. At each 100 000 clocks of timer1 interrupt routine **Timer23Int** is called and the value at port D is changed.

```
void Timer23Int() org 0x22{ // Address in the interrupt vector table of
timer3

  LATD = ~PORTD;            // Invert port D

  IFS0 = 0;                 // Clear interrupt request

}



void main(){

  TRISD = 0;                // Port D is output

  LATD  = 0xAAAA;           // Initial value at port D is set

  IPC1  = IPC1 | 0x1000;    // Timer3 priority is 1

  IEC0  = IEC0 | 0x0080;    // Timer3 interrupt enabled

  PR2   = 34464;            // Interrupt period is 100 000 clocks

  PR3   = 0x0001;           // Total PR3/2=1*65536 + 34464

  T2CON = 0x8038;           // Timer2/3 is enabled, internal clock is
divided by 256

  while(1) asm nop;         // Endless loop

}
```

How does one set timer2/3 module to the 32-bit timer mode? The corresponding iterrupt bits, i.e. interrupt priority T3IPC=1, interrupt request bit T3IF=0, and interrupt enable bit T3IE=1 are set first (in the concatenated timer2/3 module interrupts are controlled by the control bits of the timer3 module and the operation of the timer2/3 module is controlled by the control bits of the timer2 module). Then the operation of the timer is activated TON=1, 32-bit operation is selected T32=1, and in this case the prescaler is configured for 1:256 ratio TCKPS<1:0>=11. The preiod register PR2/3 contains the value 100 000 distributed according to the formula PR3/2=PR3*65536 + PR2, PR3=1 and PR2=34464.

How does one calculate the period of interrupt calls? Let the internal clock be adjusted to 10MHz. The corrsponding period is 100ns. Since the clock is divided by 256 (the prescaler reduces the clock 1:256) to form the clock of the timer, it follows that 100ns*256 = 25600ns, i.e. 25.6μs. At each 100 000 clocks an interrupt is called, i.e at each 2.56s or approximately once every two seconds.

T = 100 000*25.6μs = 2.56s.

4.4.2 32-bit synchronous counter mode

In a 32-bit synchronous counter mode the concatenated timer TMR3/TMR2 is incremented on each rising edge of the external clock signal which is synchronized with the phase of the internal clock signal of the microcontroller. When the value of the counter register TMR3/TMR2 is equal to the preset value of the PR3/PR2 register, the content of the register TMR3/TMR2 is reset to ‘0’ and an interrupt request is set by the bit T3IF.

The following example demonstrates how the timer2/3 module can be used in the 32-bit synchronous counter mode.

Example:

Use the timer2/3 for counting the external clock pulses at pin T1CK. After 10 pulses an interrupt **Timer23Int** occurs and increments the value at port D. Block diagram of connecting the timer 2/3 to the external source of clock is shown in Fig. 4-9.

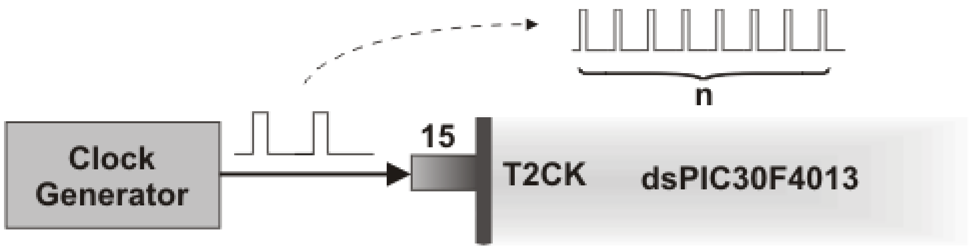


Fig. 4-9 Block diagram of connecting the timer 2/3 of a dsPIC30F4013 to an external source of clock

```
void Timer23Int() org 0x22{ //Address in the interrupt vector table of
timer3


  LATD++;                    //Increments the value of PORTD



  IFS0 = 0;                  // Clear interrupt request
}



void main(){

  TRISD = 0;              //PORTD is output

  TRISC = 0x2000;         //PORTC<13>=1 T2CK is input pin

  LATD  = 0;              //Initial value at PORTD is set

  IPC1  = IPC1 | 0x1000; //Interrupt priority of timer3 is 1

  IEC0  = IEC0 | 0x0080; //Interrupt of timer3 enabled

  PR2   = 10;             //Interrupt peiod is 10 clocks

  PR3   = 0;              //Total PR3/2=0*65536 + 10

  T2CON = 0x800A;         //Timer2/3 is synchronous counter of external
pulses

  while(1) asm nop;       //Endless loop

}
```

How does one set timer 2/3 module to the synchronous counter mode? Prescaler 1:1 is selected, external clock is enabled TCS=1, 32-bit operation is enabled T32=1, the operation of the timer1 module is enabled TON=1 in the control register T2CON (in 32-bit operation the control bits T3CON are ignored), interrupt bits of timer2/3 module are set (in the concatenated timer2/3 interrupts are controlled by the control bits of timer3 module), priority of interrupts T3IPC=1, interrupt request bit T3IF=0, and interrupt enable bit T3IE=1.

4.4.3 Gated time accumulation mode

The concatenated 32-bit timer module can be used in the gated time accumulation mode.This mode allows that the TMR3/TMR2 counter is incremented by the internal clock TCY as long as the state of the external GATE signal (pin T2CK) is high. In this way one can measure the length of the external signal. In order to operate the 32-bit timer in this mode, it is required to set the control bit TGATE (T2CON<6>) to enable this mode, select the clock source TCS=0, and enable timer operation TON=1 in the control register T2CON. In this mode the timer2 module is the internal clock source. Control bits of the timer3 T3CON are ignored.

The interrupt request is generated on the falling edge of the external GATE signal or when the value of the TMR3/TMR2 counter reaches the preset value in the PR3/PR2 register.

Attention! The falling edge of the external GATE signal does not reset the TMR3/TMR2 counter; if desired, this has to be done by the user software.

Example:

Use timer 2/3 in the gate time accumulation mode. GATE signal is applied to pin T1CK. The length of the signal is measured and displayed at pord D. Block diagram of the timer2/3 connection to an external clock sourse is shown in Fig. 4-10.

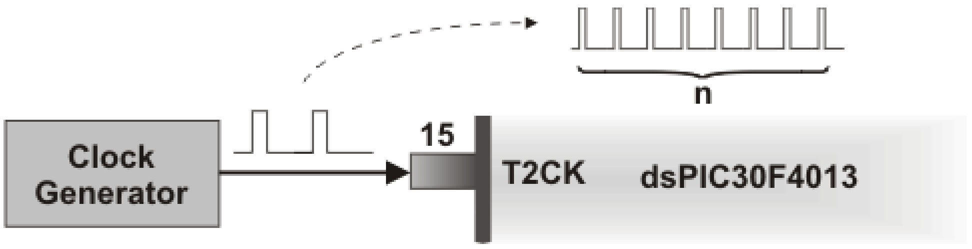


Fig. 4-10 Block diagram of the timer2/3 connection to an external clock source

```
void Timer2Int() org 0x20{ //Address in the interrupt vector table of
timer2

  LATD = TMR2;              //Signal length is displayed at port D

  IFS0.F6 = 0;              //Interrupt request cleared

}



void main(){

  T2CON = 0;                //Stops the Timer2 and reset control register

  TMR2 = 0;                 //Clear contents of the timer register

  PR2 = 0xFFFF;             //Load the Period register with 0xFFFF

  IFS0.F6 = 0;              //Interrupt request cleared

  T2CON.F6 = 1;             //Set up Timer2 for Gated time accumulation mode

  T2CON.F15 = 1;            //Start Timer2
  TRISD = 0;                //PORTD is output

  TRISC = 0x2000;           //PORTC<13>=1 is input pin

  IEC0.F6 = 1;              //Timer2 interrupt enable

  while(1) asm nop;         //Endless loop

}
```

Why the period registers PR2 and PR3 are set to the maximum value and how does one measure even longer pulses? The main reason is to measure as long pulses as possible, i.e. the interrupt does not occur because the concatenated registers TMR3/TMR2 and PR3/PR2 are equal but, if possible, it is the ceonsequence of the falling edge of the GATE signal. Measuring even longer pulses can be accomplushed by setting the prescaler to higher reduction ratios 1:8, 1:64, or 1:256. In this way the range of measurement is extended but the accuracy is reduced.

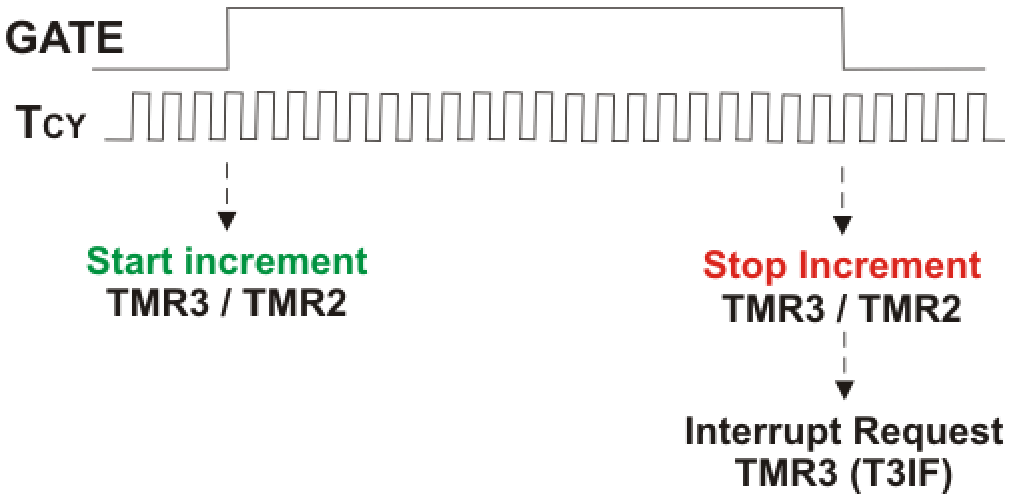

GATE
TCY
Start increment TMR3 / TMR2
Stop Increment TMR3 / TMR2
Interrupt Request TMR3 (T3IF)

Fig. 4-11 Description of the gated time accumulation mode of the 32-bit timer2/3 module

The concatenated timer2/3 has the ability to trigger an A/D conversion. This is realized by setting the corresponding control bits in the register ADCON<7:5> (SSRC<2.0>=010). When the value of the concatenated counter register TMR3/TMR2 becomes equal to that of the PR3/PR2 register, the A/D converter is triggerred and an interrupt request is generated.

NOTE: The timer modules 2 and 3 could operate independently as 16-bit timers. They coud be configured to operate in the following modes: 16-bit timer, 16-bit synchronous counter, and 16-bit gated time accumulation. The timer modules 2 and 3 can not operate as 16-bit asynchronous counters nor as real time clock sources (RTC).

As a clock source, the timer2 and 3 modules use an external clock source or the internal clock FOSC/4 with the option of selecting the prescaler reduction ratio 1:1, 1:8, 1:64, or 1:256. The selection of the reduction ratio is achieved by the control bits TCKPS<1:0> (T2CON<5:4> and T3CON<5:4>). When the timer2 and 3 modules form the concatenated 32-bit timer module, then the prescaler reduction ratio is selected by the timer2 module and the corresponding control bits of the timer3 module are ignored. The prescaler counter is reset only if: writing in the registers TMR2 or TMR3, writing in the registers PR2 or PR3, or the microcontroller is reset. It is important to note that the prescaler counter can not be reset when the timer1 module is disabled (TON=0) since the clock of the prescaler counter is stoped. Also, writing in T2CON/T3CON does not change the contents of TMR2/TMR3.

In SLEEP mode the timer2 and 3 modules are not functional since the system clock is disabled. In IDLE mode the timer2 and 3 modules will continue operation if TSIDL bit is cleared. If this bit is set, the timer2 and 3 modules will stop until the microcontroller is waken up from IDLE mode.

The characteristics of the timer4 and 5 modules are very similar to those of the timer2 and 3 modules. If concatenated to form a 32-bit timer, they could operate in the same modes as the timer2 and 3 modules. The only difference is that the timer4 and 5 modules are not used by other peripherals Iput capture and Output compare like the timer2 and 3 modules. Also, the timer5 module has no ability like the timer3 module to trigger an A/D conversion.

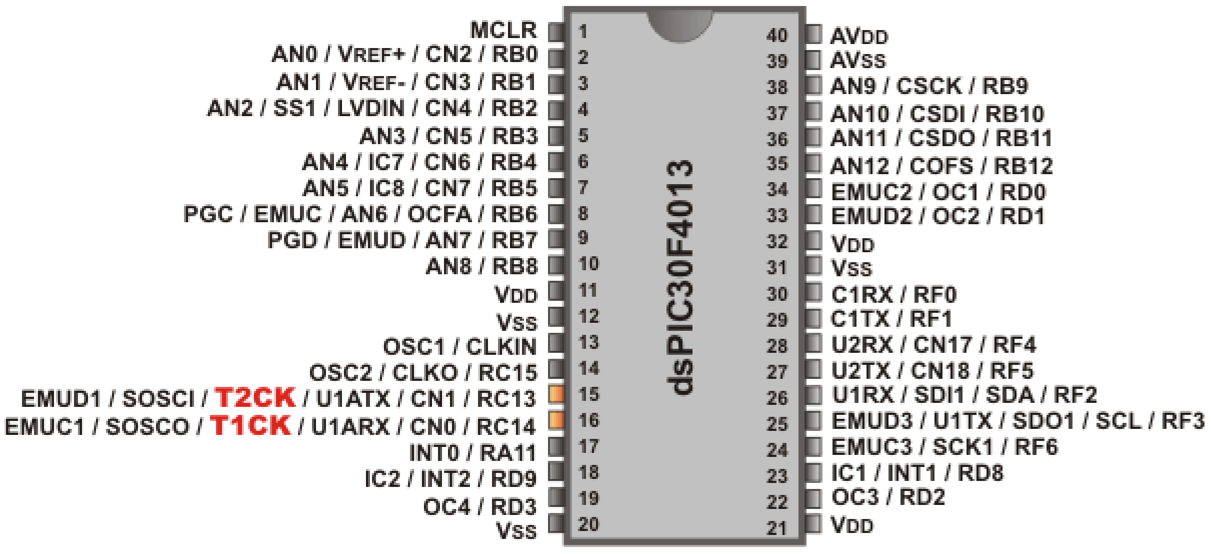


Fig. 4-12a Pin diagram of dsPIC30F4013

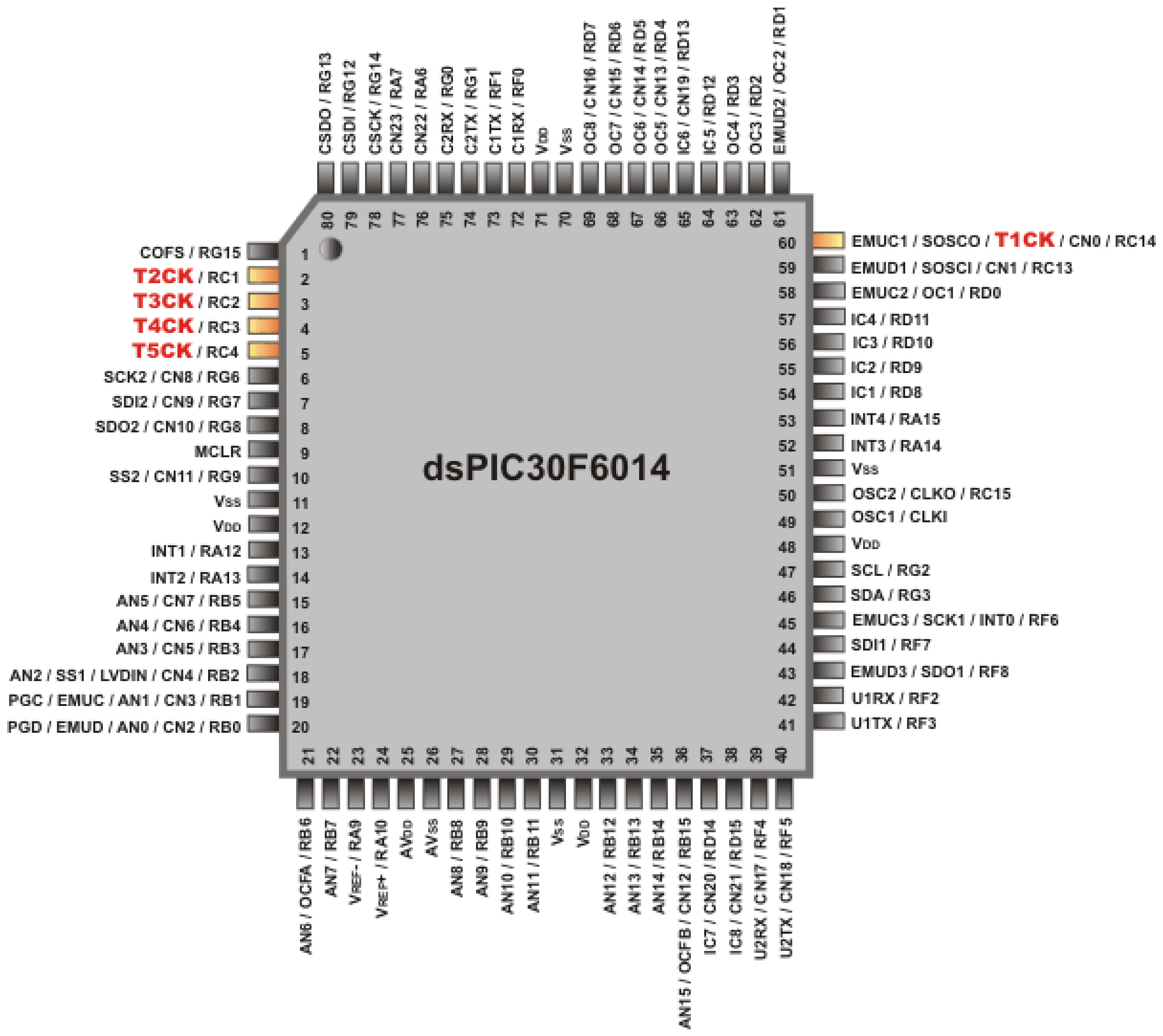


Fig. 4-12b Pin diagram of dsPIC30F6014A

Finally, a description of the registers of the timer2, 3, 4, and 5 modules of microcontroller dsPIC30F4013 is presented.

name	**ADR**	**15**	**14**	**13**	**12-7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Reset State**
TMR2	0x0106	Timer2 register											0x0000
TMR3HLD	0x0108	Timer3 holding register (32-bit operation only)											0x0000
TMR3	0x010A	Timer3 register											0x0000
PR2	0x010C	Period register 2											0xFFFF
PR3	0x010E	Period register 3											0xFFFF
T2CON	0x0110	TON	-	TSIDL	-	TGATE	TCKPS1	TCKPS0	T32	-	TCS	-	0x0000
T3CON	0x0112	TON	-	TSIDL	-	TGATE	TCKPS1	TCKPS0	-	-	TCS	-	0x0000

Table 4-2 Description of the registers of the timer2 and 3 modules

TON – Timer on control bit (TON=1 starts the timer, TON=0 stops the timer)
TSIDL – Stop in IDLE mode bit (TSIDL=1 discontinue timer operation when device enters
IDLE mode, TSIDL=0 continue timer operation in IDLE mode)
TGATE – Timer gated time accumulation enable bit (TCS must be set to 0 when TGATE=1)
TCKPS<1:0> - Timer input clock prescale select bits
00 – 1:1 prescale valu

```
      01 – 1:8 prescale value
      10 – 1:64 precale value
      11 – 1:256 prescale value
T32 – Timer 32-bit mode of timer4 and 5 select bit (T32=1 32-bit mode
selected, T32=0 timer2
      and 3 modules operate in 16-bit mode)
TCS – Timer clock source select bit
      (TCS=1 external clock from pin T1CK, TCS=0 internal clock FOSC/4)
```

NOTE: Reading unimplemented bits gives '0'.

Chapter5: Input Capture

Introduction

The input capture module has the task of capturing the curent value of the timer counter upon an input event. This module is mainly used for the frequency or time period measurements and pulse measurements (e.g. mean count rate measurement). Microcontroller dsPIC30F4013 contains 4 input capture modules, whereas dsPIC30F6014A contains 8 input capture modules.

The input capture module has multiple operatig modes selectable via the ICxCON register (control bit ICM<2:0>):

- Select by external input signal mode,
- Interrupt by external input signal mode.

The input capture module contains a four-level FIFO buffer. By setting the control bits a user can select the number of captures from the counter before the input capture module generates an interrupt request.

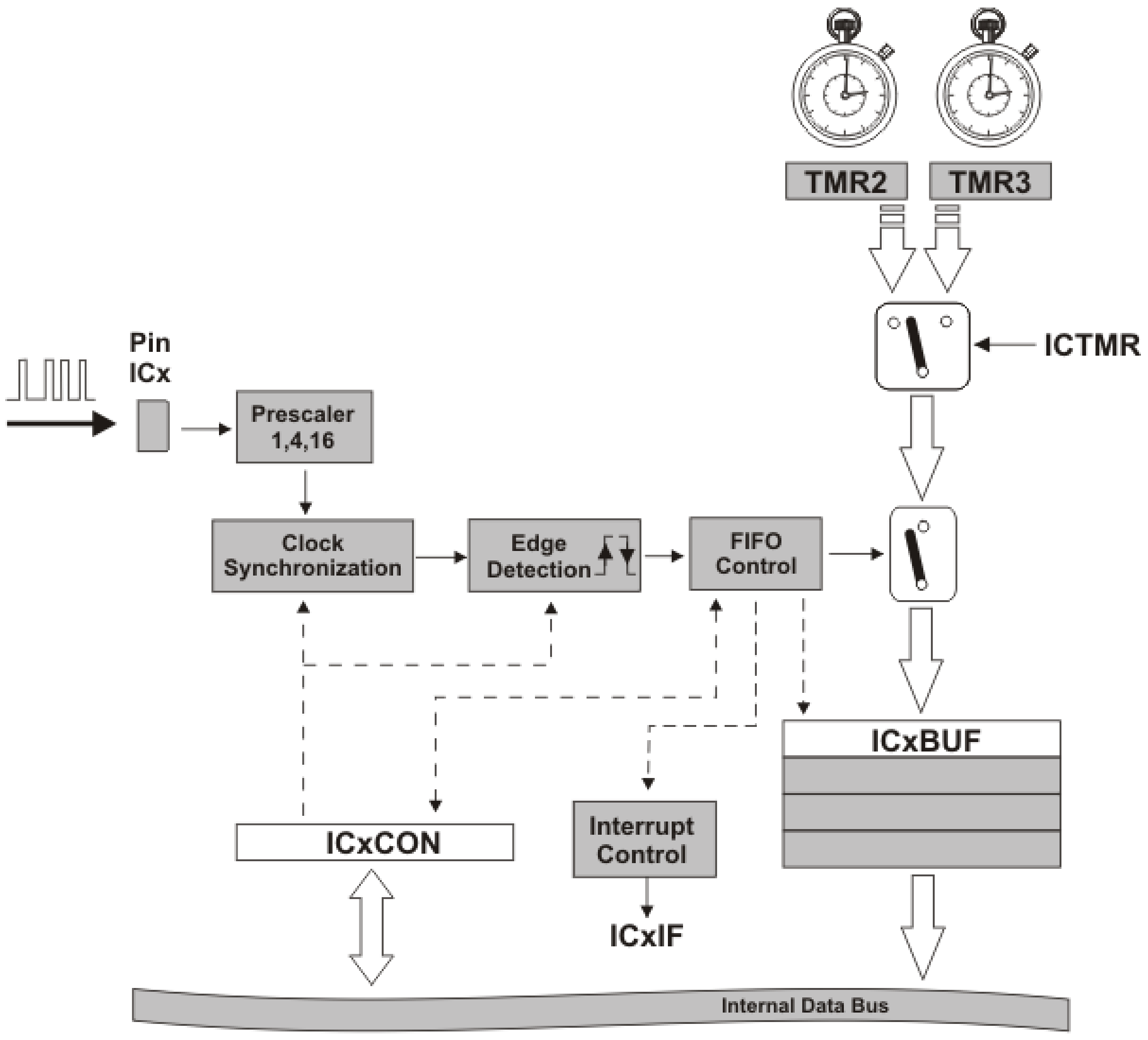


Fig. 5-1 Functional diagram of the inpuit capture module

5.1 External signal capture input mode

In the family of dsPIC30F microcontrollers the select by external input signal mode implies selecting the value from the TMR2 or TMR3 counter depending on the external input signal at pin ICx. The capture can be carried out depending on the external input signal:

- on every falling edge of input signal applied at the ICx pin,
- on every rising edge of input signal applied at the ICx pin,
- on every risinig and every falling edge of input signal applied at the ICx pin,
- on every fourth rising edge of input signal applied at the ICx pin,
- on every 16th rising edge of input signal applied at the ICx pin,

The selection of the input captue mode is carried out by setting the control bits ICM<2:0> in the register ICxCON<2:0>. Also, by setting the control bits ICM<2:0> the reduction ratio in the prescaler 1, 4 , or 16 is set.

NOTE: The counter register of the input capture module is cleared upon **RESET** or switch off.

5.1.1 Simple capture mode

The **simple capture mode**, or the mode of simple capture, is the mode of the input capture module when the capture is done on every rising edge or every falling edge of the external input signal at the input pin ICx. In this mode the logic of the input capture module detects the change of the logical level at the input pin ICx, synchronizes the change with the phase of the internal clock, captures the value of the counter TMR2 or TMR3 and puts it into the FIFO buffer memory. The prescaler operates wth the ratio 1:1, i.e. without reduction.

Since the input capture module comprises a four-level FIFO buffer, by setting the control bit ICI<1:0> (ICxCON<6:5>) it is possible to select the number of captures before an interrupt is generated. In this way capturing of fast external signals is made possible because while the counter values are captured and put into the FIFO buffer, it is possible to read previous values in the buffer and transfer them to the data memory.

Selection of the counter of the timer module which is to be captured is done by setting the control bit ICTMR (ICxCON<7>). It is possible to select the 16-bit counters TMR2 (ICTMR=1) or TMR3 (ICTMR=0).

Example:

This example demostrates the operation of the input capture module in the simple capture mode. The value of the counter of timer2 TMR2 is captured on the falling edge of the IC1 signal (pin RD8). The captured value is put to portB.

```
void Input1CaptureInt() org 0x16{

  LATB   = IC1BUF;        //Read captured values and put to portB

  IFS0.F1 = 0;            //Clear bit IC1IF (IFS<1>)

}




void main(){

  TRISB  = 0;             //PORTB is output

  LATB   = 0;             //Initial value at PORTB

  TRISD  = 0x0100;        //Select pin IC1 (RD8) as input

  IPC0   = IPC0 | 0x0010; //Interrupt priority level IC1IP<2:0> = 1

  IEC0   = IEC0 | 0x0002; //Interrupt Input Compare module enable

  PR2    = 0xFFFF;        //PR2 register at maximum, timer2 free-running
  T2CON  = 0x8030;        //Timer 2 operates with prescaler 1:256 and
internal clock

  IC1CON = 0x0082;        //Configuration of Input  Capture module 1,
selected TMR2,

                          //capture on falling edge

  while(1) asm nop;       //Endless loop

}
```

During interrupt routine clearing the interrupt Input Capture module request flag is mandatory and the captured value is read form the FIFO buffer. In setting timer2 the preset register PR2 is set at the maximum value in order to ensure operation of the timer in the free-running mode over the full range of values, from 0 to 65535. Input Capture module 1 is configured to capture values of timer 2 on falling edge of the signal at IC1 pin.

5.1.2 Prescaler capture mode

In this mode of operation of the input capture module the external signal is prescaled by the ratio 1:4 or 1:16 by setting the control bit ICM<2:0> to the values 100 or 101 respectively. In this way it is possible that the input capture module captures total value of the counter TMR2 or TMR3 for 4 or 16 periods of the external signal at the pin ICx. This is the way of measuring mean count rate by averaging 4 or 16 periods of an extarnal input signal.

By setting the control bit IC1<1:0> (ICxCON<6:5>) it is also possible, like in the simple capture mode, to select the number of captures after which an interrupt request is generated.

The selection of the timer module which is to be sampled is done by setting the control bit ICTMR (ICxCON<7>).

NOTE: If the time base is incremented by each instruction cycle, then the result of capturing will be available in the FIFO buffer one or two instruction cycles after the synchronous change at the input pin ICx, in phase with the internal clock of the microcontroller. An example of setting the captured value delayed by 1 or 2 instruction cycles TCY is shown in Fig. 5-2.

Attention!
Before the operational mode is changed the input capture module should be turned off, i.e. the control bits ICM<2:0> cleared. If the mode is changed without clearing ICM<2:0>, there is a resudual content in the prescaler counter which leads to the premature sampling and interrupt request generation.

Example:

This example demonstrates the operation of the input capture module in the prescaler capture mode. The example shows capturing the values of the timer2 counter TMR2 on each fourth rising edge if the IC1 signal (pin RD8). The captured value is put to portB.

```
void Input1CaptureInt() org 0x16{

  LATB    = IC1BUF;  //Read captured value and put to PORTB

  IFS0.F1 = 0;       //Clear IC1IF bit (IFS<1>)

}



void main(){

  TRISB  = 0;              //PORTB iz output

  LATB   = 0;              //Initial value at PORTB

  TRISD  = 0x0100;         //Select pin IC1 (RD8) as input

  IPC0   = IPC0 | 0x0010; //Interrupt priority level is

  IEC0   = IEC0 | 0x0002; //Interrupt Input Capture module 1 enable

  PR2    = 0xFFFF;         //PR2 register at maximum, timer2 free-running

  T2CON  = 0x8030;         //Timer 2 operates with prescaler 1:256 and
internal clock

  IC1CON = 0x0084;         //Configuration of Input  Capture module 1,
selected TMR2,

                           //capture on each 4th rising edge

  while(1) asm nop;        //Endless loop

}
```

During interrupt routine **clearing the interrupt Input Capture module request flag is mandatory** and the captured value is read form the FIFO buffer. In setting timer2 the preset register PR2 is set at the maximum value in order to ensure operation of the timer in the free-running mode over the full range of values, from 0 to 65535. Input Compaer module 1 is configured to capture values of timer 2 on each fourth rising edge of the signal at IC1 pin.

5.1.3 Edge detection mode

Capturing the value of TMR2 or TMR3 counter can be done on every rising and every falling edge of the external input signal applied to the ICx pin. The edge detection mode is selected by setting the ICM<2:0> (ICxCON<2:0>) control bits to 001. **In this mode the prescaler counter can not be used**. The input capture module interrupt request is generated on every rising and every falling edge (ICxIF bit is set). **It not possible to generate an interrupt request after 2, 3, or 4 captures by setting the control bits ICI<1:0> (ICxCON<6:5>) because in this mode they are ignored**. Every capture event generates an interrupt. As a consequence no overflow of the FIFO buffer is possible.

NOTE: If the time base is incremented by each instruction cycle, then the result of capturing will be available in the FIFO buffer one or two instruction cycles after the synchronous change at the input pin ICx, in phase with the internal clock of the microcontroller. An example of setting the captured value delayed by 1 or 2 instruction cycles TCY is shown in Fig. 5-2.

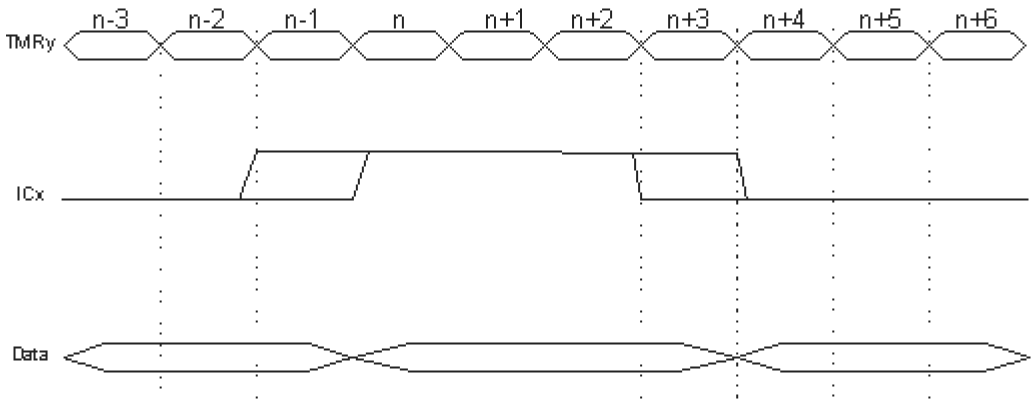


Fig. 5-2 An example of setting the captured value delayed by 1 or 2 instruction cycles TCY

Reading data from FIFO buffer – Each input capture module comprises a four-level (16-bit) FIFO buffer for accomodation of the captures. The access to the captures in the FIFO buffer is via the ICxBUF register. In addition, there are two status flags ICBNE (ICxCON<3>) and ICOV (ICxCON<4>) defining the status of the FIFO buffer. The ICBNE status flag denotes that the FIFO buffer is not empty. This flag is cleared by hardware when the last word is read from the FIFO buffer or during the reset of the input capture module by setting the control bits ICM<2:0> to value 000. ICBNE is also reset during RESET.

The other status flag ICOV denotes the state of overflow of the FIFO buffer, i.e. when after four captures which have not been transferred to the data memory the fifith capture is being put in. No interrupt request is generated then, the ICOV bit is set and the values of the five captures and all subsequent captures are ignored. Clearing of this bit is done by hardware upon reading of all four captures from the FIFO buffer, or by resetting of the input capture module. Also, the microcontroller RESET clears this flag.

5.2 External signal interrupt mode

The input pins of the input capture module can be used as additional external interrupt sources if the input capture module is configured for operation in the external signal interrupt mode. This accomplished when the configuration bits ICM<2:0> are set to 111. Then, the input pins ICx on rising edge generate an interrupt request ICxIF. If the interrupt enable bit ICxIE is set and the interrupt priority level ICxIP<2:0>is defined, the microcontroller enters an interrupt.

NOTE: Inorder that the input capture module configured to operate in the external signal interrupt mode could operate in this mode, it is necssary to clear the control bits ICI<1:0> (ICxCON<6:5>= 0). Then, the interrupt is generated independently of the reading of the FIFO buffer. Also, the status bit ICOV has to be cleared.

The input capture module is very often used by the UART module for autodetection of the baud rate. In the autobaud mode the UART module is configured as follows: the control bit ABAUD (UxMODE<5>) is set i.e. the UART RX pin is internally connected to the input capture module input. and the ICx pin is disconnected from the rest of the input capture module.The baud rate is measured by measuring the the width of the START bit when a NULL character is received. Care should be taken that the input capture module is configured for the edge detection mode. For each microcontroller of the family dsPIC30F the input capture module assignment for each UART has been defined.

5.3 Input capture operation in SLEEP and IDLE modes

Input capture operation in SLEEP mode – In SLEEP mode, the system clock is disabled, i.e. the input capture module can only function as an external interrupt source. This mode is enabled by setting control bits ICM<2:0> to 111. A rising edge on the ICx input pin will generate an input capture module interrupt. If the interrupt is enabled for this input pin, the microcontroller will wake-up from SLEEP.

In the event the input capture module has been configured for a mode other than ICM<2:0>=111 and the microcontroller enters the SLEEP mode, no external signal, rising or falling, can generate a wake-up condition from SLEEP.

Input capture operation in IDLE mode – Operation of the input capture module in IDLE mode is specified by the control bit ICSIDL (ICxCON<13>). If ICSIDL= 0, the module will continue operation in IDLE mode with full functionality in all the above mentioned modes. The prescaler is fully functional in this mode.

If ICSIDL=1, the module will stop in IDLE mode. Then, the input capture module can operate only in the external signal interrupt mode, i.e. the control bits ICM<2:0>=111. A rising edge on the ICx input pin will generate an input capture module interrupt. If the interrupt is enabled for this input pin, the microcontroller will wake-up from IDLE state.

In the event the input capture module has been configured for a different mode and the microcontroller enters the IDLE mode with the control bit ICSIDL is set, no external signal, rising or falling, can generate a wake-up condition from IDLE state.

Attention!
When the input capture module is enabled, a user software has to enable that the ICx input pin is configured as the input pin by setting the associated TRIS bit. By enabling the input capture

module the pin ICx is not configured automatically. Also, all other peripherals using this pin have to be disabled.

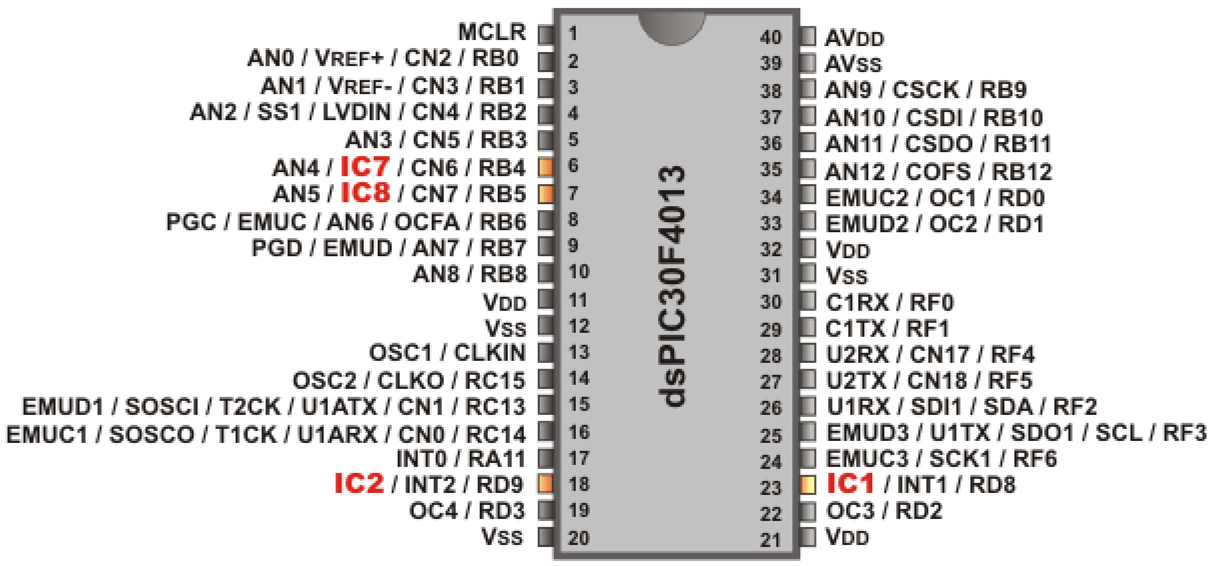


Fig. 5-3a Pin diagram of dsPIC30F4013

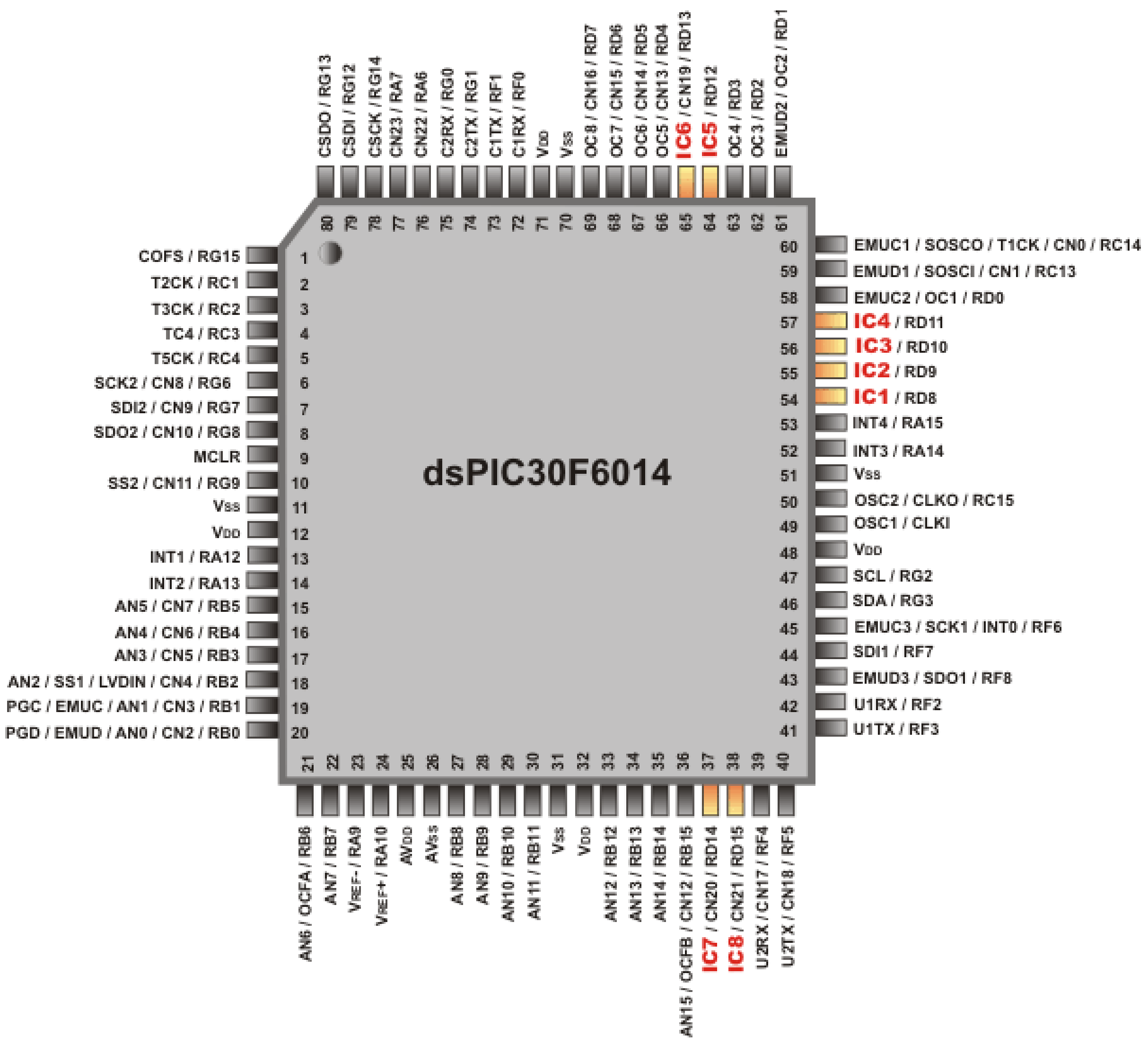


Fig. 5-3b Pin diagram of dsPIC30F6014A

Finally, a description of the registers of the input capture module of microcontroller dsPIC30F4013 is presented.

name	ADR	15	14	13	12-8	7	6	5	4	3	2	1	0	Reset State
IC1BUF	0x0140	Input 1 Capture Buffer Register												0xuuuu
IC1CON	0x0142	-	-	ICSIDL	-	ICTMR	ICI<1:0>		ICOV	ICBNE	ICM<2:0>			0x0000
IC2BUF	0x0144	Input 2 Capture Buffer Register												0xuuuu
IC2CON	0x0146	-	-	ICSIDL	-	ICTMR	ICI<1:0>		ICOV	ICBNE	ICM<2:0>			0x0000
IC7BUF	0x0158	Input 7 Capture Buffer Register												0xuuuu
IC7CON	0x015A	-	-	ICSIDL	-	ICTMR	ICI<1:0>		ICOV	ICBNE	ICM<2:0>			0x0000
IC8BUF	0x015C	Input 8 Capture Buffer Register												0xuuuu
IC8CON	0x015E	-	-	ICSIDL	-	ICTMR	ICI<1:0>		ICOV	ICBNE	ICM<2:0>			0x0000

Table 5-1 Input capture module registers

```
ICSIDL – Input captur module stop in IDLE control bit
        (ICSIDL=0 input capture module will continue to operate in IDLE
mode,
        ICSIDL=1 input capture module will halt in IDLE mode)

ICTMR – Input capture timer select bits (ICTMR=0 TMR3 contents are captured
on capture event,
        ICTMR=1 TMR2 contents are captured on capture event)

ICI <1:0> - Select number of captures per interrupt bits
      00 – interrupt on every capture event
      01 – interrupt on every second capture event
      10 – interrupt on every third capture event
      11 – interrupt on every fourth capture event

ICOV – FIFO buffer overflow status flag (read only) bit

ICBNE – FIFO buffer buffer empty status (read only) bit
        (ICBNE=0 FIFO buffer empty, ICBNE=1 FIFO buffer contains at least
one capture value

ICM <2:0> - Input capture mode select bits
      000 – Input capture module turned off
      001 – Capture mode, every edge (rising or falling)
      010 – Capture mode, every falling edge
      011 – Capture mode, every rising edge
      100 – Capture mode, every 4th rising edge
      101 – Capture mode, every 16th rising edge
      110 – Unused (module disabled)
      111 – Input capture module in external signal interrupt mode
      (external source of interrupt requests)
```

Chapter6: Output Compare Module

Introduction

The output compare module has the task of comparing the value of the time base counter with the value of one or two compare registers depending on the Operation mode selected. It is able to generate a single output pulse or a sequence of output pulses when the compared values match; also, it has the ability to generate interrupts on compare match events.

The dsPIC30F4013 controller has 4 output compare modules whereas controller dsPIC6014A has 8. Each output compare channel can select which of the time base counters, TMR2 or TMR3, will be compared with the compare registers. The counter is selected by using control bit OCTSEL (OCxCON<3>).

The output compare module has several modes of operation selectable by using control bits OCM<2:0> (ocXcon<2:0>):

- Single compare match mode,
- Dual compare match mode generating either one output pulse or a sequence of output pulses,
- Pulse Width Modulation (PWM) mode.

NOTE: It is advisable, before switching to a new mode, to turn off the output compare module by clearing control bit OCM<2:0>.

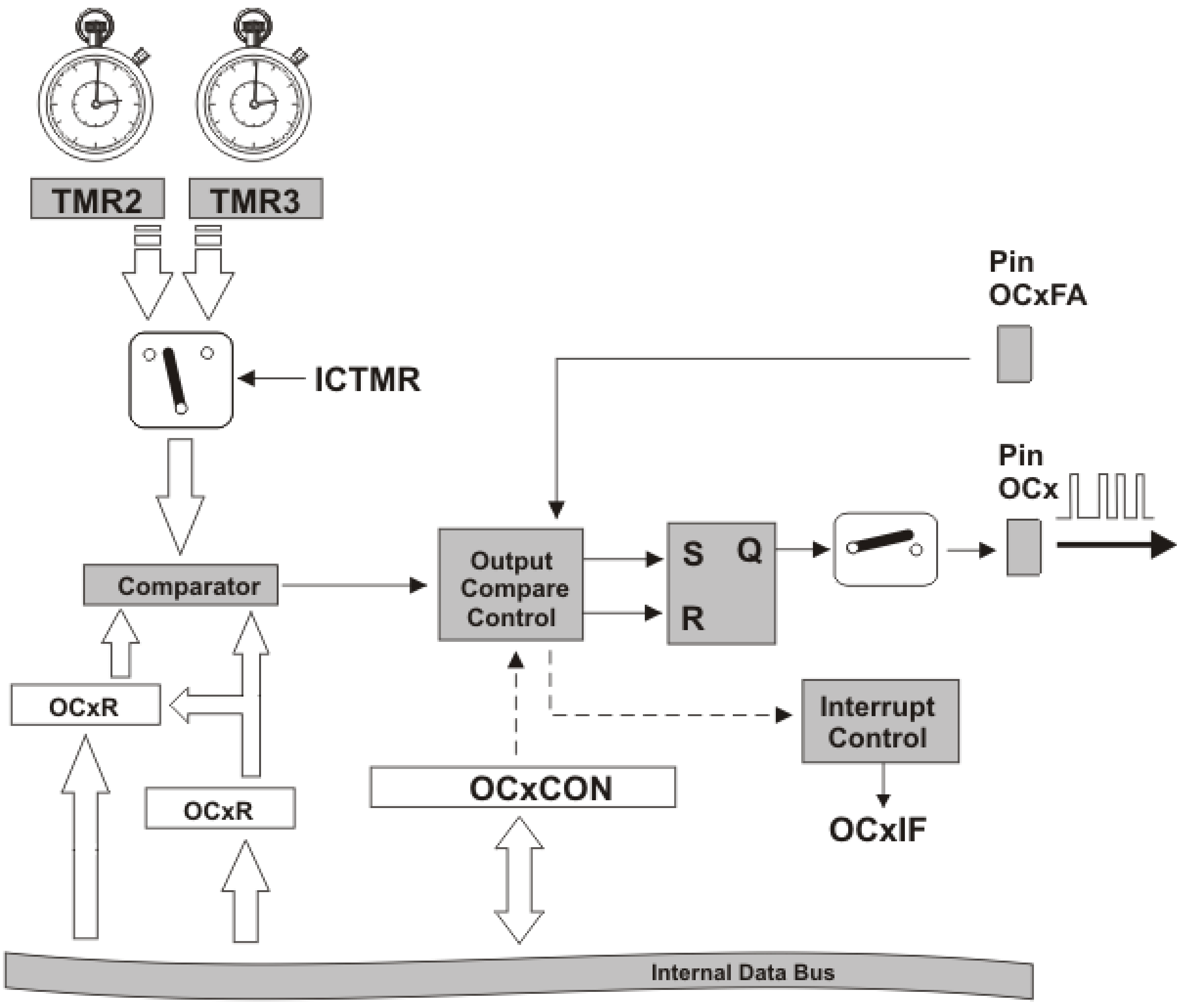


Fig. 6-1 Functional diagram of output compare module

6.1 Single compare match mode

When control bits OCM<2:0> are set to 001, 010, or 011, the ouput compare module is set to the Single compare match mode. Now, the value loaded in the compare register OCxR is compared with time base counter TMR2 or TMR3. On a compare match event, depending on the value of OCM<2:0>, at the OCx output pin one of the following situations is possible:

- OCx pin is high, initial state is low, and interrupt is generated,
- OCx pin is low, initial state is high, and interrupt is generated,
- State of OCx pin toggles and interrupt is generated.

6.1.1 Single compare match, pin OCx driven high

In order to configure the output compare module for this mode, control bits OCM<2:0> are set to 001. Also, the time base counter (TMR2 or TMR3) should be selected. Initially, output pin OCx is set low and will stay low until a match event occurs between the TMRy and OCxR registers. One instruction clock after the compare match event, OCx pin is driven high and will remain high until a change of the mode or the module is disabled. TMRy goes on counting. Twop instruction clocks after OCx pin is driven high, the interrupt, OCxIF flag, is generated. Timing diagram of the single compare mode, set OCx high on compare match event is shown in Fig. 6-2.

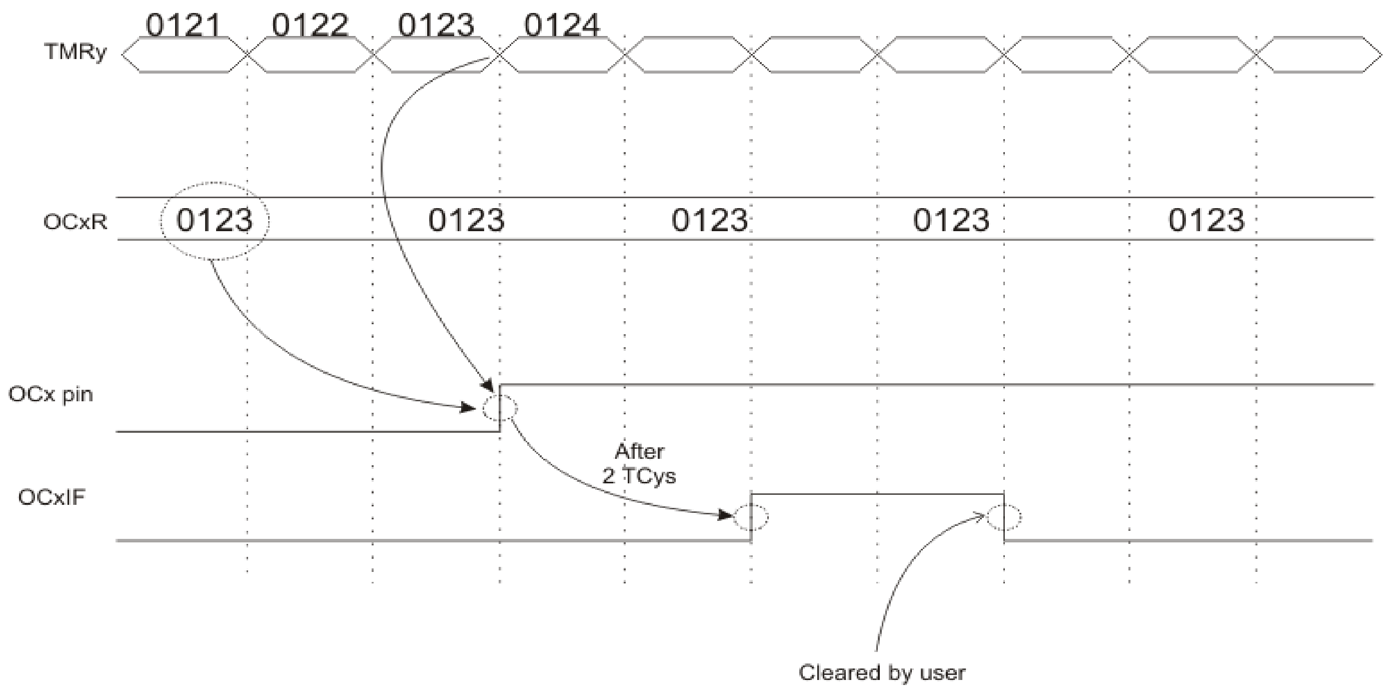


Fig. 6-2 Timing diagram of the single compare mode, set OCx high on compare match event

6.1.2 Single compare match, pin OCx driven low

In order to configure the output compare module for this mode, control bits OCM<2:0> are set to 010. Also, the time base counter (TMR2 or TMR3) should be enabled. Initially, output pin OCx is set high and it stays high until a match event occurs between the TMRy and OCxR registers. One instruction clock after the compare match event OCx pin is driven low and will remain low until a change of the mode or the module is disabled. TMRy goes on counting. Two instruction clocks after OCx pin is driven low, the interrupt flag, OCxIF, is generated. Timing diagram of the single compare mode, set OCx low on compare match event is shown in Fig. 6-3.

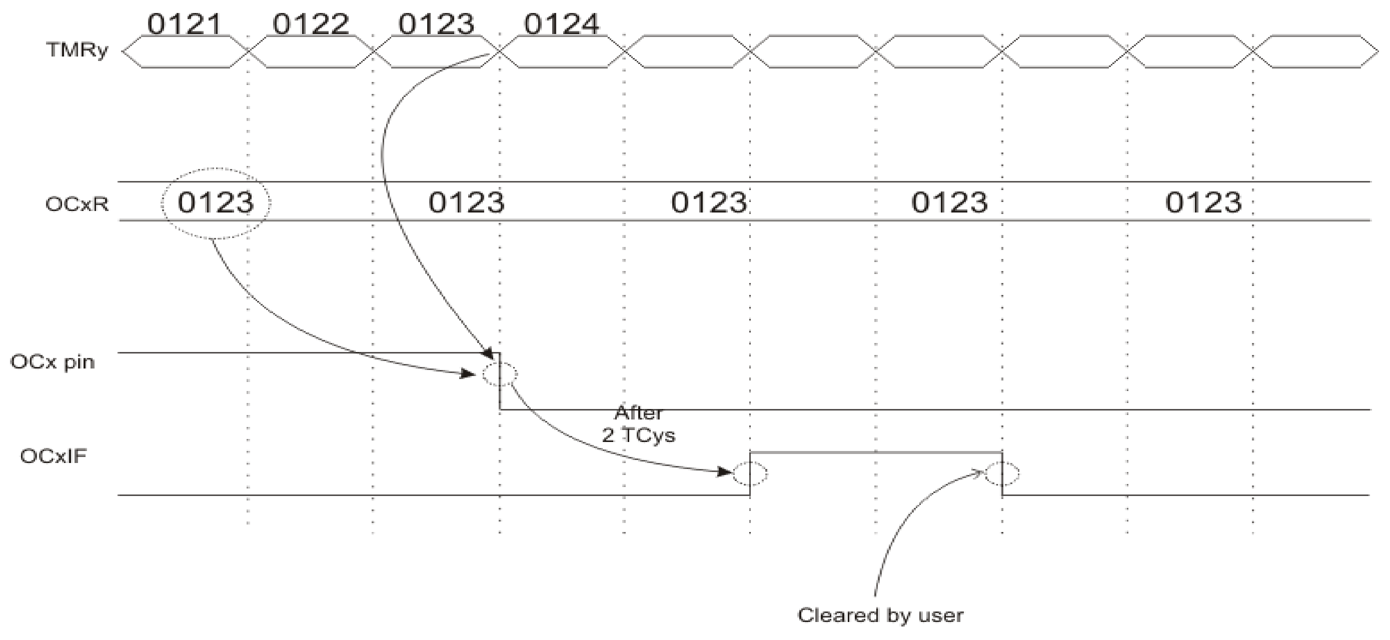


Fig. 6-3 Timing diagram of the single compare mode,set OCx low on compare match event

6.1.3 Single compare match, pin OCx toggles

In order to configure the output compare module for this mode, control bits OCM<2:0> areset to 011. Also, the time base counter (TMR2 or TMR3) should be enabled. Initially, output pin OCx is set low and then toggles on each subsequent match event between the TMRy and OCxR registers. OCX pin is toggled one instruction clock the compare match event. TMRy goes on counting. Two instruction clocksafter the OCX pin is togglrd, the interrupt flag, OCxF, is generated. Figs. 6-4 and 6-5 show the timing diagrams of the single compare mode, toggle output on compare match event when timer register PRy (PR2 or PR3)>OCxR (Fig. 6-4) or timer register PRy (PR2 or PR3)=OCxR (Fig. 6-5).

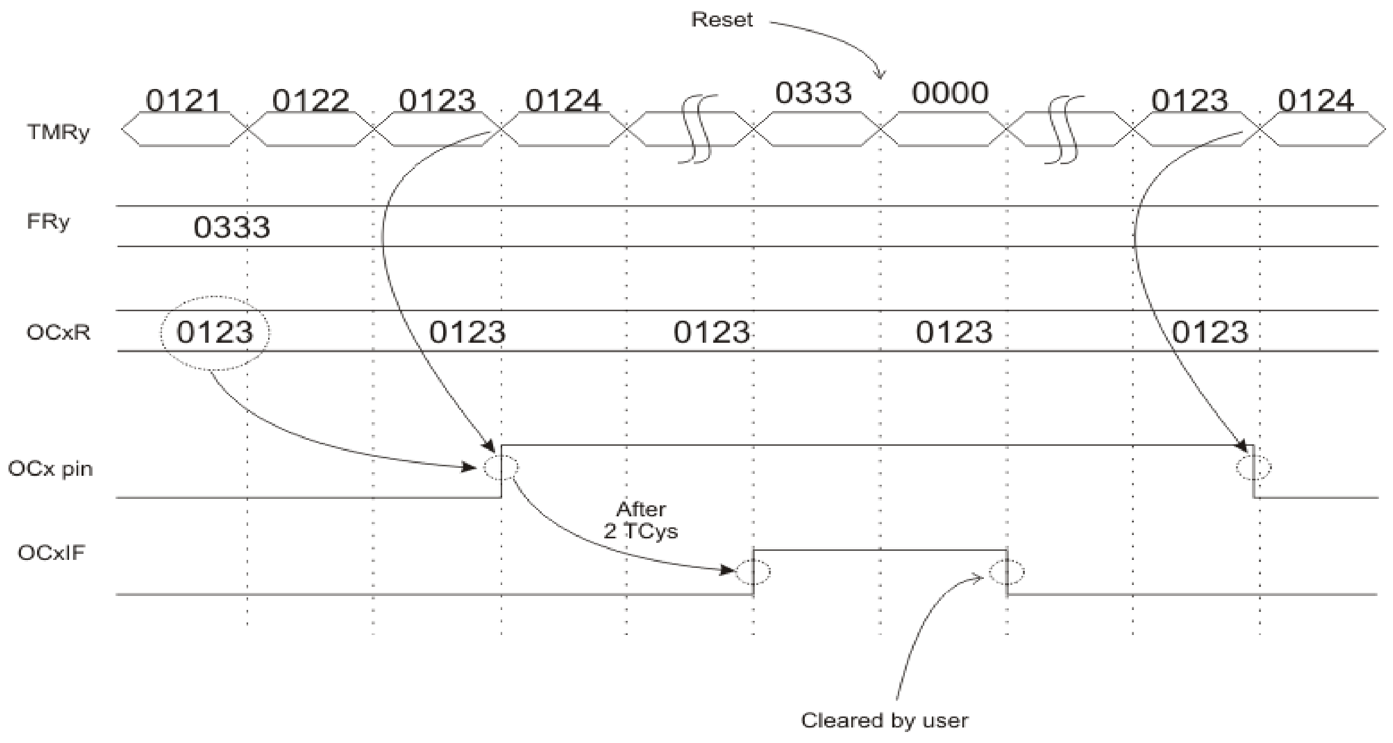


Fig. 6-4 Timing diagrams of the single compare mode, toggle output on compare match event when timer register PRy>OCxR

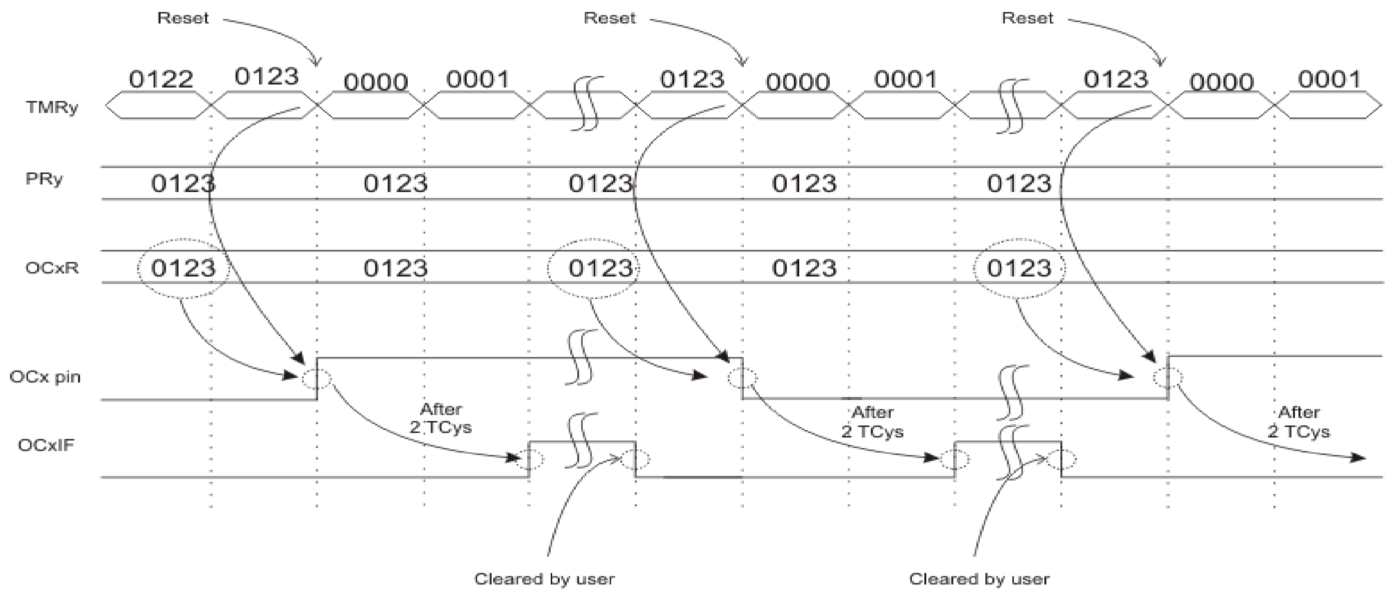


Fig. 6-5 Timing diagrams of the single compare mode, toggle output on compare match event when timer register PRy=OCxR

NOTE: OCx pin is set low on a device RESET. In the single compare mode, toggle output on compare match event the operational OCx pin state can be set by the user software.

Example:

Output compare module 1 is in the single compare mode: toggle current output of pin OC1. The output compare module compares the values of OC1R and TMR2 registers; on equality, the output of OC1 pin is toggled

```
void Output1CompareInt() org 0x18{ //OC1 address in the interrupt vector
table

  IFS0.F2 = 0;                         //Clear Interrupt Flag

}



void main(){

  TRISD  = 0;              //OC1 (RD0) is output pin

  IPC0   = IPC0 | 0x0100; //Priority level of interrupt OC1IP<2:0>=1

  IEC0   = IEC0 | 0x0004; //Output compare 1 enable interrupt

  OC1R   = 10000;         //OCR=TMR2 instant of level change at OC1

  PR2    = 0xFFFF;        //PR2 value maximal, time base 2 free-running

  T2CON  = 0x8030;        //Time base 2 operates using prescaler 1:256 and
internal clock

  OC1CON = 0x0003;        //Output compare 1 module configuration,TMR2
selected

                          //Single compare mode, pin OC1 toggles

  while(1) asm nop;       //Endless loop

}
```

In the interrupt routine the request for the flag Output compare interrupt module is reset. At setting time base 2, preset register PR2 is set to the maximum value in orde to enable the free-running mode over the whole range, 0-65335. The value of OC1R defines the time of the change of state of pin OC1, i.e. of the duty cycle. The output compare module is configured to change the state of pin OC1 on single compare match with the value of OC1R.

6.2 Dual compare match mode

When control bits OCM<2:0> are set to 100 or 101, the output compare module is configured for the dual compare match mode. In this mode the module uses two registers, OCxR and OCxRS, for the compare match events.

The values of both registers are compared with the time base counter TMR2 or TMR3. On a compare match event of the OCxR register and register TMR2 or TMR3 (selectable by control bit OCTSEL), the leading edge of the pulse is generated at the OCx pin; the register OCxRS is then compared with the same time base register and on a compare match evenet, the trailing edge at the OCx pin is generated.

Depending on the value of control bit OCM<2:0> at the output pin OCx is generated:

- single pulse and an interrupt request,
- a sequence of pulses and an interrupt request.

6.2.1 Dual compare match mode, single output pulse at pin OCx

When control bits OCM<2:0> are set to 100, the output compare module is configured for the dual compare match (OCxR and OCxRS registers), single output pulse mode. By setting the control bits OCTSEL the time base counter for comparison is selected. v. Two instruction clocks after pin OCx is driven low, an interrupt request OCxIF for the output compare module is generated. Pin OCx will remain low until a mode change has been made or the module is disabled. If the contents of time base register PRy<OCxRS, no trailing edge and the interrupt request are generated and if PRy<OCxR no leading edge is generated at pin OCx. Fig. 6-6 shows timing diagram of the operation of the output compare module in the dual compare match mode, single pulse at the output pin OCx (OCxR < OCxRS < PRy).

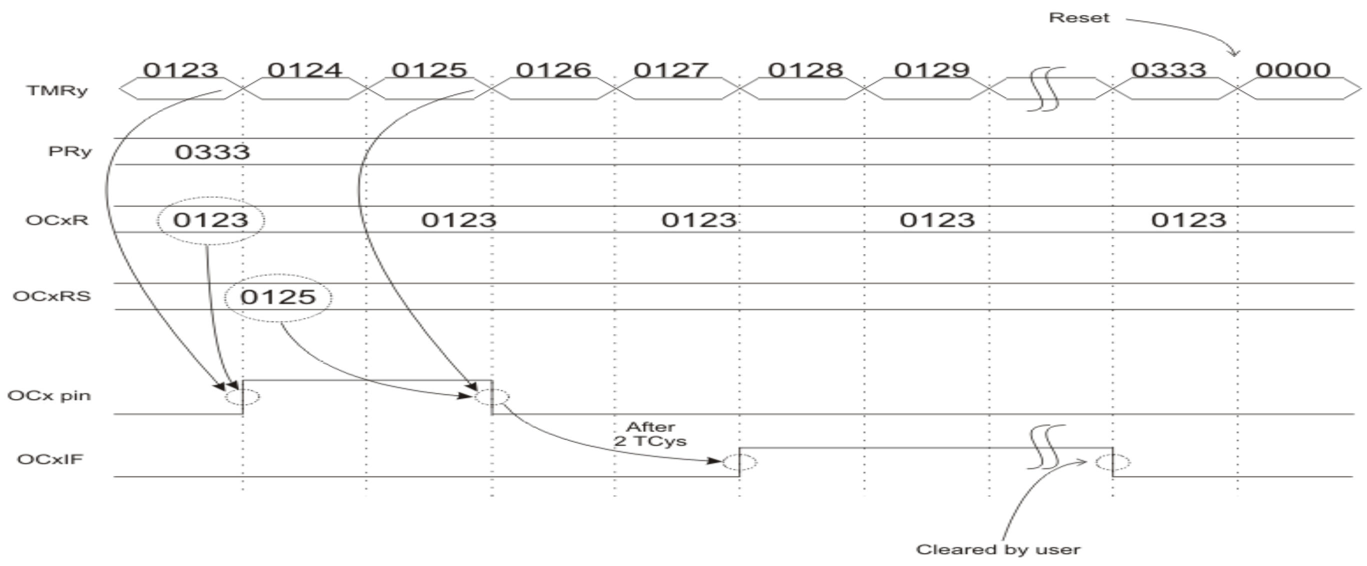


Fig. 6-6 Timing diagram of the operation of the output compare module in the dual compare match mode, single pulse at the output pin OCx

6.2.2 Dual compare match mode, sequence of output pulses at pin OCx

When control bits OCM<2:0> are set to 101 the output compare module is configured for the dual compare match (OCxR and OCxRS registers), a sequence of output pulses is generated at the output OCx pin. After a compare match occurs between the compare time base (TMR2 or TMR3) and OCxR registers, the output pin OCx is driven high, i.e. the leading edge is generated at pin OCx. When a compare match occurs between the time base (TMR2 or TMR3) and OCxRS registers, the trailing edge at pin OCx is generated, i.e. pin OCx is driven low. Two instruction clocks after, an interrupt request for the otput compare module is generated. In this mode **it is not necessary to reset the ouput compare module** in order that the module could react on equalization of the TRy and OCxR or OCxRS registers. Even if the interrupt of the output compare module is enabled (OCxIE is set), it is required that in **the interrupt routine the interrupt request flag of the output compare module OCxIF is reset**.

If the preset register PRy < OCxRS, the trailing edge at pin OCx will not be generated. Fig. 6-7 shows an example of operation of the output compare module in the dual compare match mode, a sequence of output pulses at the output pin OCx (OCxR < OCxRS < PRy).

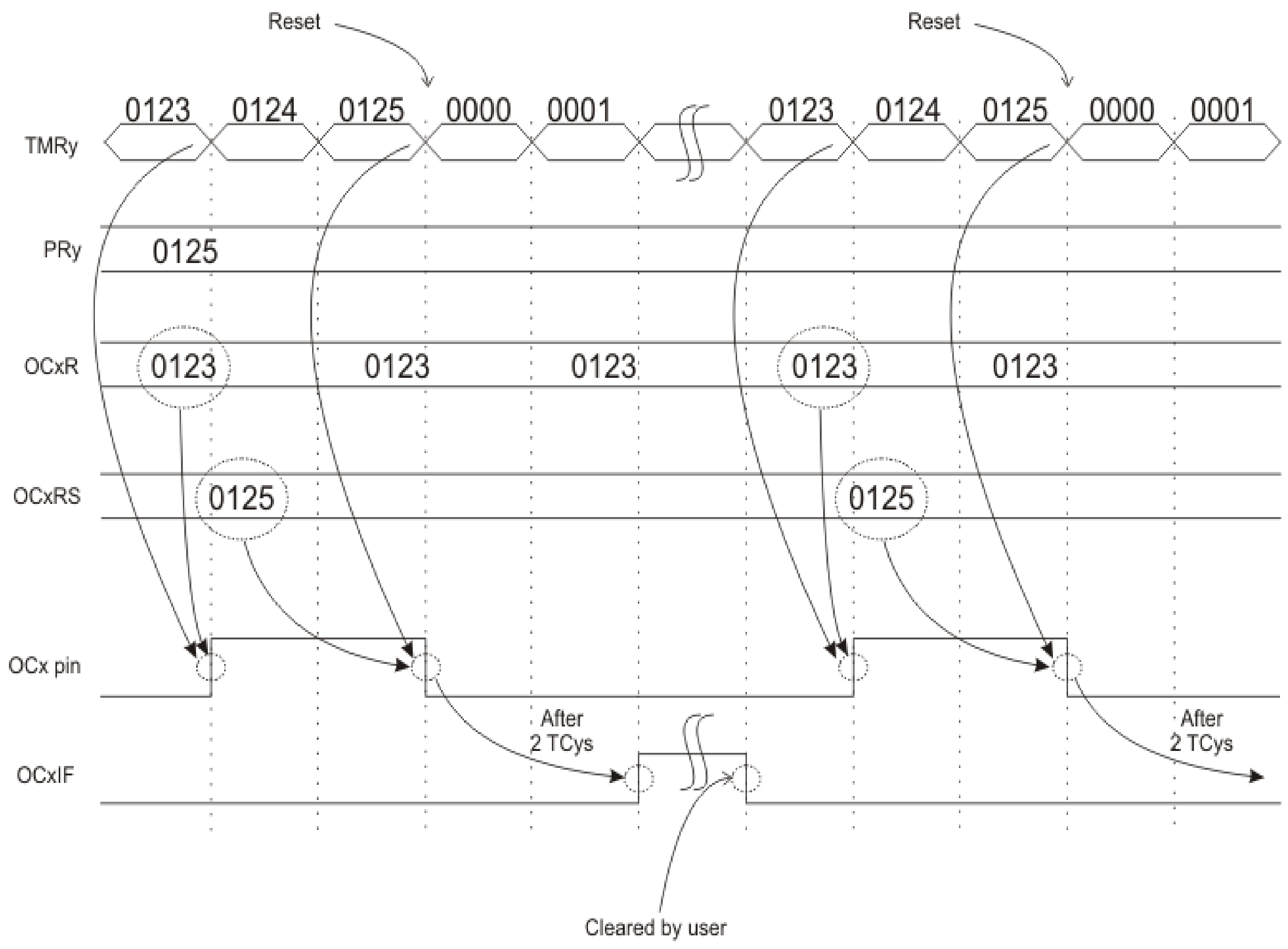


Fig. 6-7 Timing diagram of the operation in the dual compare match mode, pulse sequence at the output pin OCx

Example:

The output compare module compares the value of registers OC1R and OC1RS with the value of the counter TMR2 of the time base 2 and on compare match event toggles the logical level at pin OC1.

```
void Output1CompareInt() org 0x18{//Address of OC1 in the interrupt table

 IFS0 = 0;   //Reseting of interrupt OC1 module flag

}



void main(){

  TRISD = IPC0 | 0x0100; //OC1 (RD0) is output

  IPC0  = IEC0 | 0x0004; //Output  Compare module 1 interrupt enable

  OC1R  = 30000;      //If OC1R=TMR2, leading edge at pin OC1

  OC1RS = 100;        //If OC1RS=TMR2, trailing edge at OC1

  PR2   = 0xFFFF;     //PR2 at maximum, time base 2 free-running

  T2CON = 0x8030;     //Time base 2 operates with prescaler 1:256 and
internal clock

  OC1CON = 0x0005;    //Configuration of Output Compare 1 module,

                      //TMR2 selected, dual compare match, pulse sequence

  while(1) asm nop;   // Endless loop

}
```

In the interrupt routine **the interrupt request flag of the Output Compare module is reset**. In presetting timer 2, register PR2 is set to the maximum value in order to enable free-running mode of the timer within the entire range of values 0 to 65535. The value of OC1R defines the instant of the leading edge at pin OC1, the value of OC1RS defines the instant of the trailing edge. The Output Compare module is configured to toggle continually the logical level at pin OC1 on dual compare match event with the values of registers OC1R and OC1RS.

6.3 The Pulse Width Modulation (PWM) Mode

When control bits OCM<2:0> are set to the values 110 or 111, the output compare module is configured for the pulse width modulation (PWM) mode. The PWM mode is available without fault protection input or with fault protection input. For the second PWM mode the OCxFA or OCxFB input pin is used. Fig. 6-8 shows an example of microcontroller dsPIC30F4013 connection to the inverter including the feedback error signal. The inverter controls the operation of motor M

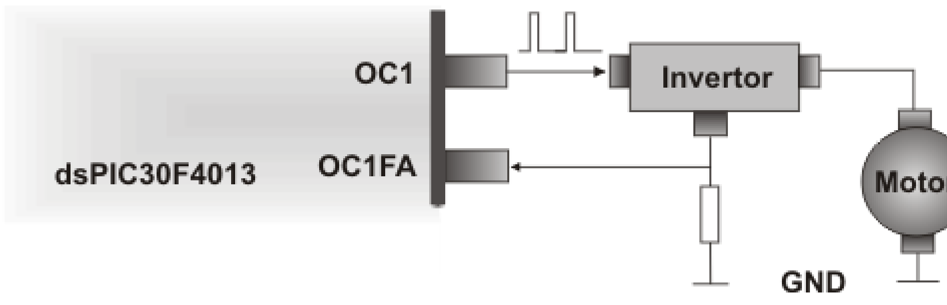


Fig. 6-8 dsPIC30F4013 connection to the inverter including the feedback error signal

6.3.1 PWM mode without fault protection input

When control bits OCM<2:0> are set to 110, the output compare module operates in this mode. In PWM mode the OCxR register is a read only slave duty cycle register. The OCxRS is a buffer register written by the user to update the PWM duty cycle.On every timer to period register match event (end of PWM period), the duty cycle register, OCxR, is loaded with the contents of OCxRS.The interrupt flag is asserted at the end of each PWM period.

When configuring the output compare module for PWM mode of operation, the following steps should be taken:

1. Set the PWM period by writing to the selected timer period register, PRy.
2. Set the PWM duty cycle by writing to the OCxRS register.
3. Write the OCxR register with the initial duty cycle.
4. Enable interrupts for the selected timer.
5. Configure the output compare module for one of two PWM operation modes by writing 100 to control bits OCM<2:0> (OCxCON<2:0>).
6. Set the TMRy prescale value and enable the selected time base.

NOTE: The OCxR register should become a read only duty cycle register before the output compare module is first enabled (OCM<2:0>=000).

6.3.2 PWM mode with fault protection input pin

When the control bits OCM<2:0> areset to 111, the outpu compare module is configured for the PWMmode of operation. All functions derscribed in section 6.3.1 apply, with the addition that in this mode in addition to the output pin OCX the use is made of the signal from the input pin OCxFA for the output compare channels 1 to 4 or from the inpit pin OCxFB for the output compare channels 5 to 8. The signal at input pin OCxFA or OCxFB is a feedback error signal of the inverter related to a possible hazardous state of operation of the inverter. If the input pin OCxFA or OCxFB is low, the inverter is onsidered to be in a hazardous (error) state. Then the output OCx pin of the output compare module operating in the PWM mode is disabled automatically and the pin is driven to the high impedance state. The user may elect to provide a pull-down or pull-up resistor in order to define the state of OCx pin which is in thisstate disconnected from the rest of the output compare module. In the state of inverter fault, upon detection of the fault condition and disabling of pin OCx, the respective interrupt flag is asserted and in the register OCxCON the OCFLT bit (OCxCON<4>) is set. If enabled, an interrupt of the output compare module will be generated.

NOTE: The external fault pins, OCxFA or OCxFB, while the output compare module operates in PWM mode with fault protection input pin, will continue to protect the module while it is in SLEEP or IDLE mode.

6.4 PWM period and duty cycle calulation

6.4.1 PWM period

The PWM period, specified by the value in the PRy register of the selected timer y, is calculated by:

TPWM=(PRy +1)TCY(TMRy prescale value),

and the PWM frequencyby:

fPWM=1/TPWM.

Example:

Calculation of the PWM period for a microcontroller having a 10MHz clock with x4 PLL, Device clock rate is 40MHz. The instruction clock frequency is FCY=FOSC/4, i.e. 10MHz. Timer 2 prescale setting is 4. Calculate the PWM period for the maximum value PR2=0xFFFF=65535.

TPWM = (65535+1) x 0.1µs x (4) = 26.21 ms, i.e. fPWM = 1/TPWM = 38.14 Hz.

6.4.2 PWM duty cycle

The PWM duty cycle is specified by the value written to the register OCxRS. It can be written to at any time within the PWM cycle, but the duty cycle value is latched into OCxR when the PWM period is completed. OCxR is a read only register. This provides a double buffering for the PWM duty cycle.

If the duty cycle register,OCxR, is loaded with 0000, the duty cycle is zero and pin OCx will remain low throughout the PWM period.

If the duty cycle register is greater that PRy, the output pin OCx will remain high throughout the PWM period (100% duty cycle).

If OCxR is equal to PRy, the OCx pin will be high in the first PWM cycle and low in the subsequent PWM cycle.

NOTE: In order to achieve as fine as possible control of PWM, it is necessary to enable as high as possible duty cycle adjustment. This is accomplished by adjusting the prescale value, clock value, and PWM frequency to achieve the highest possible value of PRy thus achieving the highest number of adjustment levels, i.e. the highest resolution.

6.4.3 Maximum resolution

The maximum resolution is calculated by the following formula:

Max PWM resolution [bits] = (log10TPWM - log(TCY x prescale value TMRy)) / log102

Example:

Calculation of the PWM period for a microcontroller having a 10MHz clock with x4 PLL, Device clock rate is 40MHz. The instruction clock frequency is FCY=FOSC/4, i.e. 10MHz. Timer 2 prescale setting is 4. Calculate the maximum resolution for PWM frequency 48Hz.

Max PWM resolution [bits] = (log10(1/48) – log10(0.1µs x 4)) / log102 = 15.66 bits.

For this value of the PWM period and other selected parameters (prescale value, clock) it turns out that the PWM mode operates with almost maximum resolution.

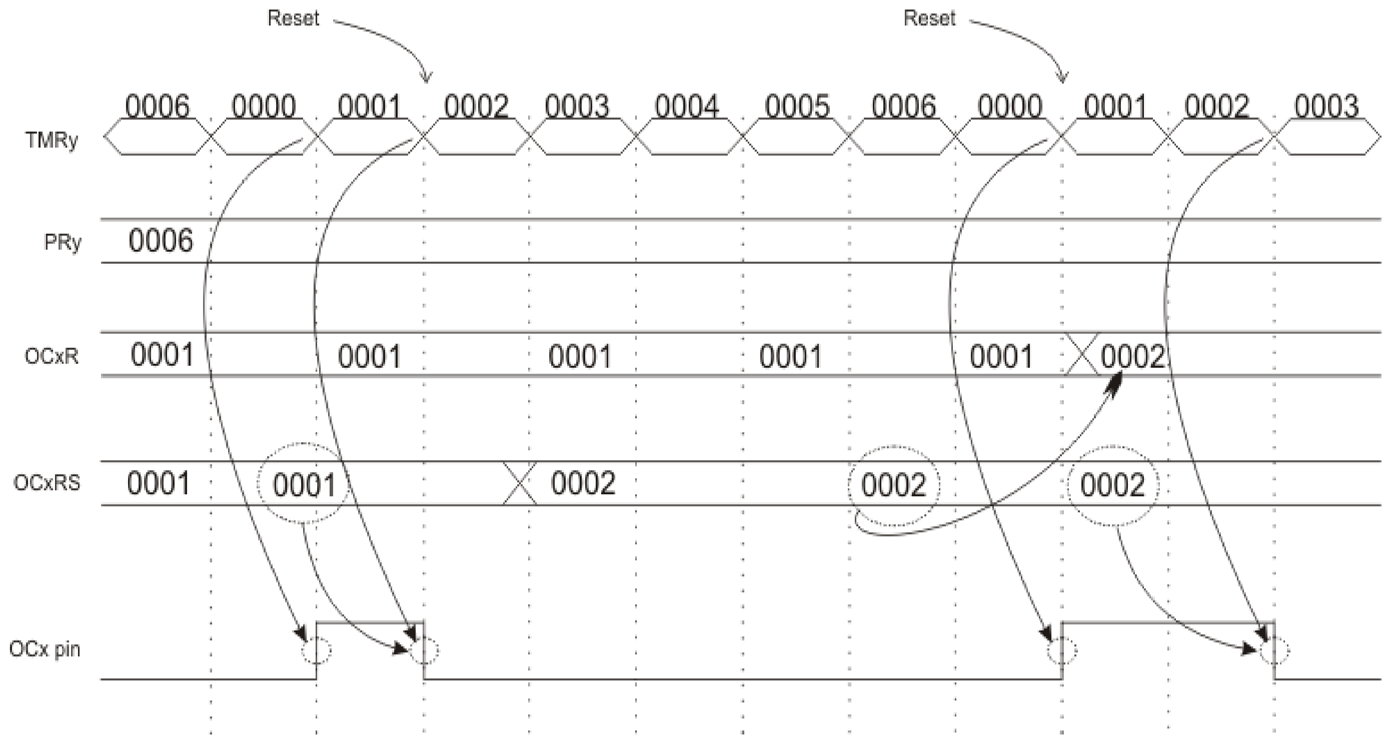


Fig. 6-9 Timing diagram of the PWM mode of the output compare module

6.5 Operation of the output compare module in SLEEP or IDLE mode

6.5.1 Operation of the output compare module in SLEEP mode

When the device enters SLEEP mode, the system clock is disabled. During SLEEP, the state of the output pin OCx is held at the same level as prior to entering SLEEP. For example, if the output pin OCx was high and the CPU entered the SLEEP state, the pin will stay high until the the microcontroller wakes up.

6.5.2 Operation of the output compare module in SLEEP mode

When the device enters IDLE mode, the OCSIDL control bit (OCxCON<13>) selects if the output compare module will stop or continue operation in IDLE mode.

If OCSIDL = 0, the module will continue operation only if if the selected time base is set to operate in IDLE mode (TSIDL = 0).

If OCSIDL = 1, the module will discontinue operation in IDLE mode as it does for SLEEP mode.

NOTE: The input pins OCxFA and OCxFB during SLEEP or IDLE modes, if enabled for use, will continue to control the associated output pin OCx, i.e. disconnect the output pin OCx if they are low.

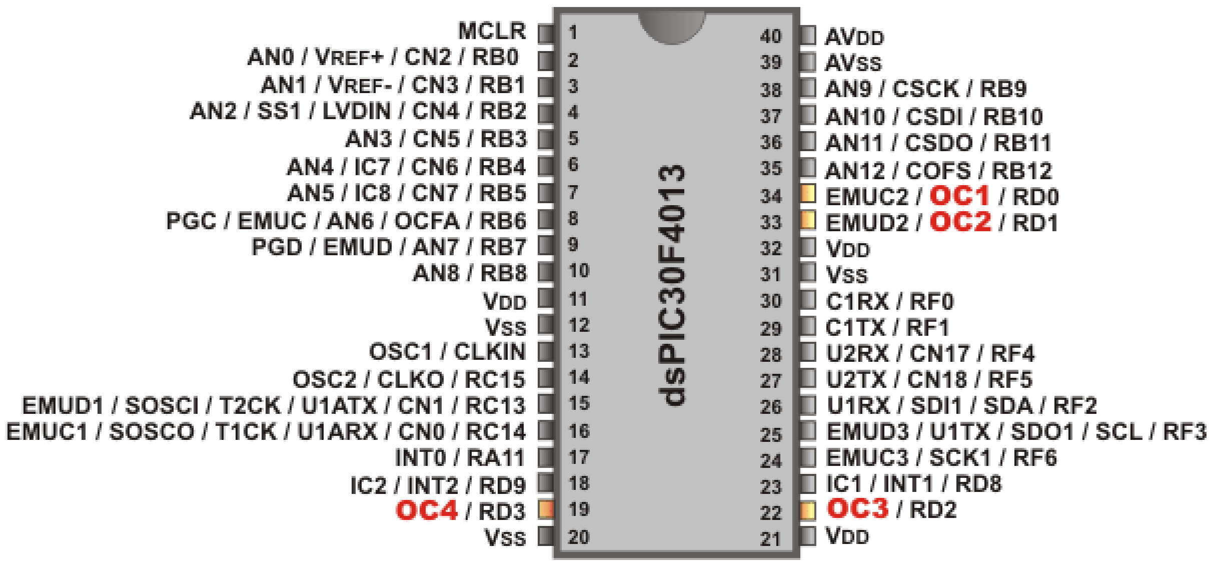


Fig. 6-10a Pinout of microcontroller dsPIC30F4013, output compare module pins marked

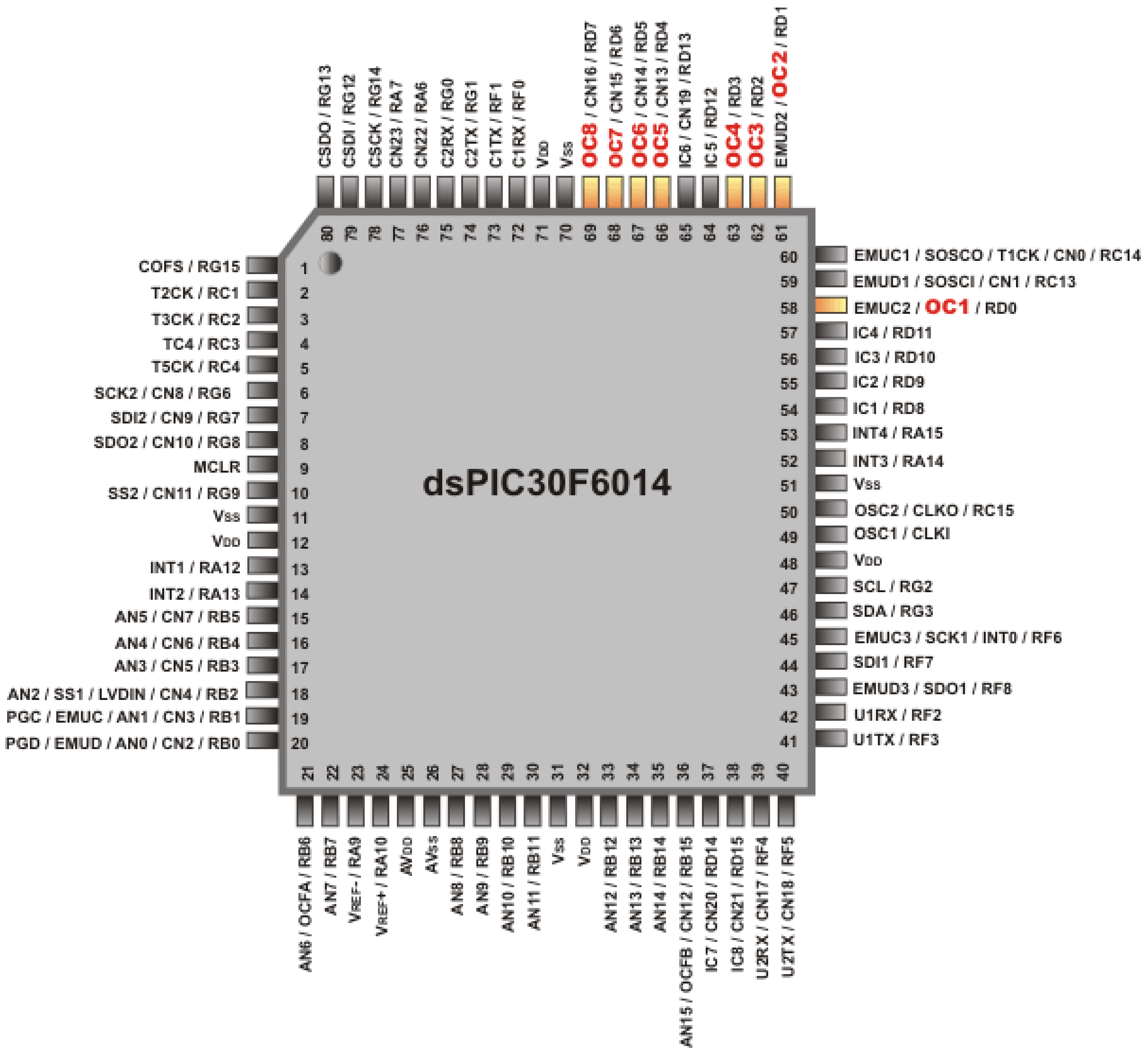


Fig. 6-10b Pinout of microcontroller dsPIC30F6014A, output compare module pins marked

Finally, a description is given of the output registers of the output compare module of microcontroller dsPIC30F4013.

name	ADR	15	14	13	12-5	4	3	2	1	0	Reset State
OC1RS	0x0180	Output Compare 1 Secondary Register									0x0000
OC1R	0x0182	Output Compare 1 Main Register									0x0000
OC1CON	0x0184	-	-	OCSIDL	-	OCFLT	OCTSEL	OCM<2:0>			0x0000
OC2RS	0x0186	Output Compare 2 Secondary Register									0x0000
OC2R	0x0188	Output Compare 2 Main Register									0x0000
OC2CON	0x018A	-	-	OCSIDL	-	OCFLT	OCTSEL	OCM<2:0>			0x0000
OC3RS	0x018C	Output Compare 3 Secondary Register									0x0000
OC2R	0x018E	Output Compare 3 Main Register									0x0000
OC3CON	0x0190	-	-	OCSIDL	-	OCFLT	OCTSEL	OCM<2:0>			0x0000
OC4RS	0x0192	Output Compare 4 Secondary Register									0x0000
OC4R	0x0194	Output Compare 4 Main Register									0x0000
OC4CON	0x0196	-	-	OCSIDL	-	OCFLT	OCTSEL	OCM<2:0>			0x0000

Table 6-1 Register map associated with output compare module

```
OCSIDL – output compare stop bit in IDLE state (OCSIDL=0 the module is
active in
         IDLE state, OCSIDL=1 the module in inactive in IDLE state)
OCFLT – PWM FAULT state bit (OCFLT=0 no FAULT occured, OCFLT=1 FAULT
        occured, hardware reset only)
OCTSEL – Output Compare timer select bit (OCTSEL=0 TMR2 selected, OCTSEL=1
         TMR3 selected)
OCM <2:0> - mode select bit of the Output Compare Module
     000 – Output Compare Module disabled
     001 - Single compare match mode, pin OCx driven high
     010 - Single compare match mode, pin OCx driven low
     011 -  Single compare match mode, pin OCx toggles
     100 - Dual compare match mode, single output pulse at pin OCx
     101 - Dual compare match mode, sequence of output pulses at pin OCx
     110 - PWM mode without fault protection input
     111 - PWM mode with fault protection input
```

Chapter7: A/D Converter

Introduction

A/D (Analogue-to-Digital) converter is a “mixed signal” circuit which performs digitization of the external analogue signals. In the dsPIC30F family there are two microcontroller versions one with 10-bit and the other with 12-bit A/D converter. It has been shown in practice that for the control or regulation systems the 10-bit A/D converters gave satisfactory solutions. The additional bits in the A/D converters in these applications are almost unusable because they are masked by the high levels of electro-magnetic noise. In the measuring systems the practice showed that with the 12-bit converters one could achieve a good compromise between the sampling rate and accuracy (0.02% for a 12-bit converter).

The 10-bit A/D converter microcontrollers could sample up to 16 analogue inputs with the frequency of 500kHz applying the successive approximation (SAR) conversion. The successive approximation conversion method is one of the basic types of A/D conversion. It is carried out in several steps until the input voltage is equal to the internally generated voltage. The speed of conversion of this type of A/D converters is limited, but they are sufficiently fast for most of the general purpose applications. This 10-bit A/D converter has the ability of using external voltage references, simultaneous sampling of up to four analogue inputs in the Sample/Hold amplifiers (**sampling of several signals at one time is possible**), automatic channel scan mode, and selectable conversion trigger source. The conversion result is loaded to a 16-word buffer register in four selectable output formats (two integer and two floating point). In addition, the A/D converter could operate in SLEEP or IDLE mode.

The performance of microcontrollers having 12-bit A/D converter is somewhat inferior to this, but their conversion accuracy is higher. They could sample up to 16 analogue inputs with 100kHz frequency applying the successive approximation conversion. The converter can use the external voltage references, sample only one of the 16 analogue inputs in the Sample/Hold amplifier, it has automatic channel scan mode and selectable conversion trigger source. Like the 10-bit converter, the conversion result is loaded to a 16-word buffer register in four selectable output formats (two integer and two floating point). In addition, the A/D converter could operate in SLEEP or IDLE mode.

7.1 12-bit A/D converter

Functional block diagram of the 12-bit A/D converter used in the dsPIC30F4013 microcontroller is shown in Fig. 7-1.

The A/D converter possesses 12 analogue inputs (other microcontrollers of the dsPIC30F family could have up to 16 analogue inputs) connected via an analogue multiplexer to the Sample/Hold amplifier. There are two analogue inputs for setting an external reference voltage. The output of the Sample/Hold amplifier is the input to the converter, based on successive approximations. The 12-bit A/D converter can be configured that the reference voltages are the supply voltages (AVDD, AVSS) or external reference voltages (VREF+, VREF-). This A/D converter can operate in SLEEP mode which is very useful for the purpose of minimizing conversion noise owing to the power supply. For controlling the process of A/D conversion the A/D converter has six 16-bit control registers: registers ADCON1, ADCON2, ADCON3 serving for selecting the mode of the A/D converter, register ADCHS serving for selecting analogue inputs used for A/D conversion, register ADPCFG serving for selecting the pin used as analogue input and the pin used as Input/Output (I/O) pin, and register ADCSSL serving for selecting the analogue inputs to be scanned.

The result of A/D conversion is loaded in a «read-only» RAM buffer, 16-word deep and 12-bit wide. The result of conversion from the buffer denoted by ADBUF0, ADBUF1, ... ADBUFF is, via the output 16-bit register, read in accordance with the selected output format. The content of the RAM buffer at locations ADBUF0 to ADBUFF can not be written by software, but is written exclusively by the A/D converter.

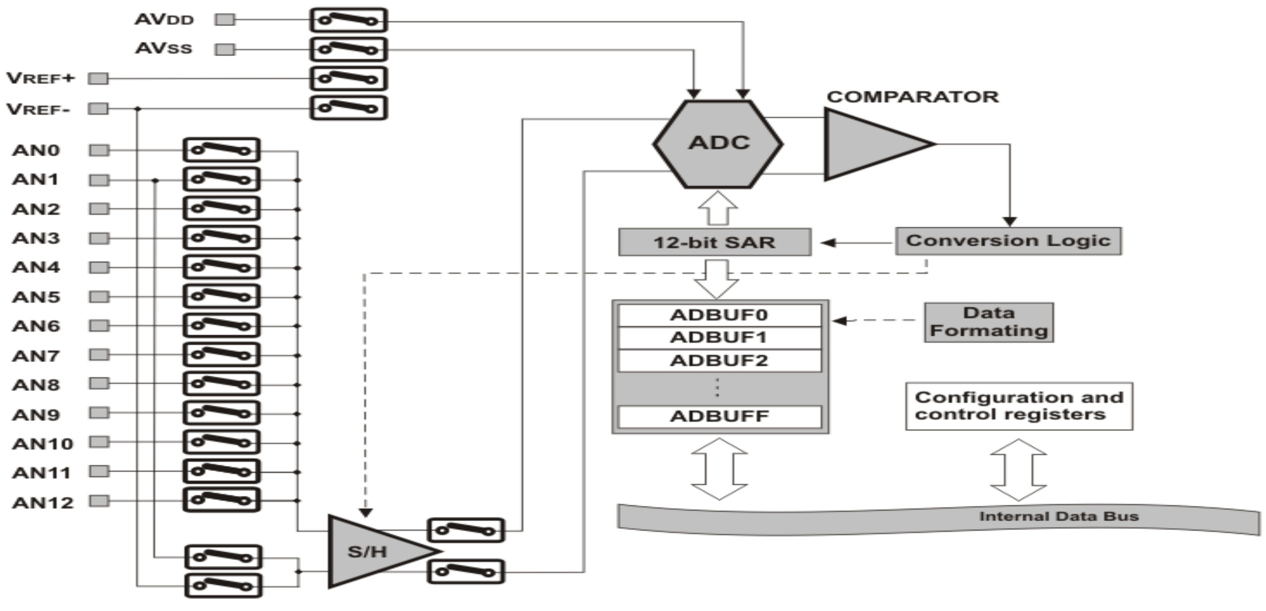


Fig. 7-1 Functional block diagram of the 12-bit A/D converter of the dsPIC30F4013 device

Attention!
The control bits SSRC<2:0> and ASAM (register ADCON1), SMPI<3:9>, BUFM and ALTS (register ADCON2), as well as registers ADCON3 and ADCON4 must not be written during A/D conversion. By writing during A/D conversion the flow of the process is changed and the result is not controlled. If the bit ADON in the ADCON1 register is set to logic zero during A/D conversion, the conversion is interrupted. The correponding partial results are not written in the correspopnding ADBUFn register.

7.1.1 A/D conversion sequence

Upon configuration of the A/D converter, the process of acquisition of samples of the input signalis is started by setting the SAMP bit. The conversion may be started by various sources: programmable control bit, timer after a preset time interval, or an external event. When the conversion time is complete, the result is loaded into the RAM buffer located from ADBUF0 to ADBUFF registers. Upon completion of the A/D conversion, the bit DONE and interrupt flag are set after the number of samples defined by the SMPI control bits.

The following steps should be followed for performing an A/D conversion:

1. Configure the A/D module
   - configure the port pins as analogue inputs, voltage reference, and digital I/O pins,
   - select A/D converter input channel,
   - select A/D conversion clock,
   - select A/D conversion trigger source,
   - turn on A/D module;
2. Configure A/D interrupt (if required)
   - clear ADIF bit (IFS0,11>),
   - select A/D interrupt priority,
   - set ADIE bit (IEC0<11>);
3. Start sampling
4. Wait the required acquisition time;
5. Trigger acquisition end, start conversion;
6. Trigger acquisition end, start conversion;
7. Wait for A/D to complete, by either
   - waiting for the A/D interrupt, or
   - waiting for the DONE bit to get set;
8. Read A/D result buffer, clear ADIF bit if required.

7.1.2 A/D converter configuration

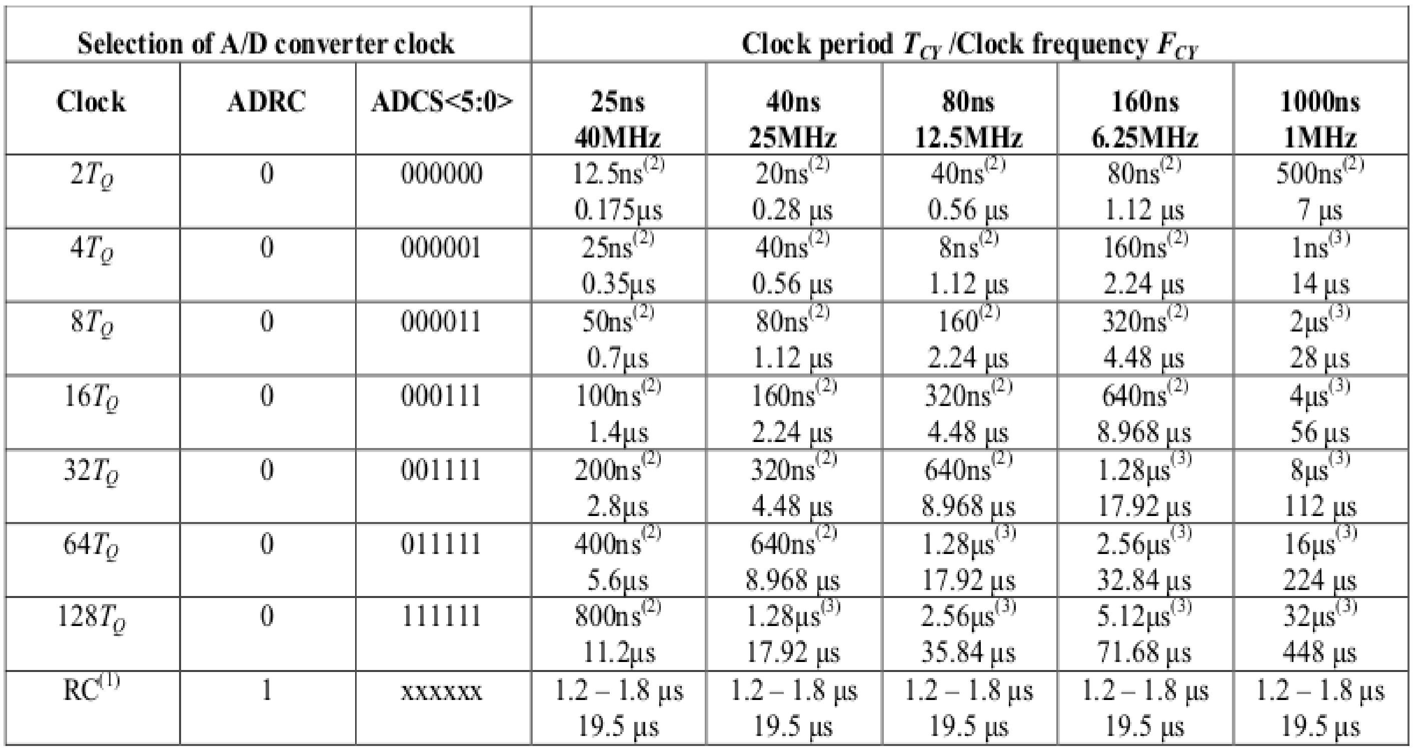
Select voltage reference – A/D converter configuration is started by selecting the voltage reference source. This process is performed by setting the control bit VCFG<2:0> (ADCON2<15:13>). The reference voltage could be selected to be either internal supply voltages AVDD or AVSS or external voltage references VREF+ or VREF- via the external pins.

Select the A/D conversion clock – A/D conversion rate is determined by the A/D conversion clock, the period is denoted by TAD . For performing one A/D conversion 14 periods of the TAD clock are required. The A/D converer clock is formed on the basis of the instruction clock TCY of the microcontroller. The A/D conversion period TAD , generated by a 6-bit counter, is selected by the control bits ADCS<5:0> (ADCON3<5:0>). Period TAD is defined by the formula:

$$T_{AD} = \frac{T_{CY}(ADCS+1)}{2},$$

$$ADCS = \frac{2T_{AD}}{T_{CY}} - 1.$$

In order that the A/D conversion gives a correct result, it is required that the conversion time, 14TAD , is at least 10µs. From this condition one can conclude that the minimum period of the A/D converter clock is TAD = 667ns (applicable to dsPIC30F3014 and dsPIC30F6014A). For the 10-bit A/D converters of the dsPIC30F family the minimum period of the clock is 154ns. Table 7-1 presents the values of the period of the A/D converter clock and the length of A/D conversion as functions of the instruction clock TCY of the micocontroller.



Selection of A/D converter clock			Clock period T_{CY} /Clock frequency F_{CY}				
Clock	ADRC	ADCS<5:0>	25ns 40MHz	40ns 25MHz	80ns 12.5MHz	160ns 6.25MHz	1000ns 1MHz
$2T_Q$	0	000000	12.5ns(2) 0.175µs	20ns(2) 0.28 µs	40ns(2) 0.56 µs	80ns(2) 1.12 µs	500ns(2) 7 µs
$4T_Q$	0	000001	25ns(2) 0.35µs	40ns(2) 0.56 µs	8ns(2) 1.12 µs	160ns(2) 2.24 µs	1ns(3) 14 µs
$8T_Q$	0	000011	50ns(2) 0.7µs	80ns(2) 1.12 µs	160(2) 2.24 µs	320ns(2) 4.48 µs	2µs(3) 28 µs
$16T_Q$	0	000111	100ns(2) 1.4µs	160ns(2) 2.24 µs	320ns(2) 4.48 µs	640ns(2) 8.968 µs	4µs(3) 56 µs
$32T_Q$	0	001111	200ns(2) 2.8µs	320ns(2) 4.48 µs	640ns(2) 8.968 µs	1.28µs(3) 17.92 µs	8µs(3) 112 µs
$64T_Q$	0	011111	400ns(2) 5.6µs	640ns(2) 8.968 µs	1.28µs(3) 17.92 µs	2.56µs(3) 32.84 µs	16µs(3) 224 µs
$128T_Q$	0	111111	800ns(2) 11.2µs	1.28µs(3) 17.92 µs	2.56µs(3) 35.84 µs	5.12µs(3) 71.68 µs	32µs(3) 448 µs
RC(1)	1	xxxxxx	1.2 – 1.8 µs 19.5 µs	1.2 – 1.8 µs 19.5 µs	1.2 – 1.8 µs 19.5 µs	1.2 – 1.8 µs 19.5 µs	1.2 – 1.8 µs 19.5 µs

Table 7-1 The values of the A/D converter clock period and length of A/D conversion TAD as functions of the instruction clock period TCY of the micocontroller

NOTE:
(1) If the clock source is the internal RC oscillator, typical values are TAD=1.5ns and VDD > 3V.
(2) The value is less than the minimum value of TAD.
(3) If shorter conversion time is desired, selection of another clock source is recommended.
(4) A/D converter does not operate with full accuracy if the internal RC clock source operates beyond 20MHz.

Selection of analogue inputs – All Sample/Hold amplifiers have analogue multiplexers for selecting analogue input pins determined by the control bits ADCHS. The control bits ADCHS determine which analogue inputs are selected for each sample (non-inverting and inverting).

Configuring analogue port pins – The ADPCFG register specifies the input condition of device pins used as analogue inputs. A pin is configured as an analogue input when the corresponding PCFGn bit (ADPCFG<n>) is cleared. If the corresponding PCFGn is set, the pin is configured as a digital I/O. The ADPCFG register is cleared at RESET, causing the A/D input pins to be configured for analogue inputs. The TRISB register determines whether a digital port is input or output. If the corrsponding TRIS bit is set, the pin is input, if this bit is cleared, the pin is output. Configuration of the ADPCFG and TRISB registers controls the operation of the A/D port pins.

Channel 0 input selection – The user may select any of the up to 16 analogue inputs as the input to the positive input of the Sample/Hold amplifier by setting the control bits CH0SA<3:0> (ADCHS<3:0>). The user may select either VREF- or A1 as the negative input of the channel, depending on the setting the CH0NA bit. In this way channel 0 is fully defined.

Specifying altenating channel 0 input selections – The ALTS control bit causes the module to alternate between two sets of inputs that are selected during successive samples. The inputs specified by CH0SA<3:0>, CH0NA, CHXSA, and CHXNA<1:0> are collectively called the MUX A inputs. The inputs specified by CH0SB<3:0>, CH0NB, CHXSB, and CHXNB<1:0> are collectively called the MUX B inputs.

When the ALTS control bit is set, then one input of the group MUX A and one input of the group MUX B are selected alternatively. When this control bit is cleared, only the group MUX A inputs are selected for sampling.

Scanning through several inputs with channel 0 – Channel 0 has the ability to scan through a selected vector of inputs. The CSCNA bit (ADCON2<10>) enables the CH0 channel inputs to be scanned across a selected number of analogue inputs. When CSCNA bit is set, the CH0SA<3:0> bits are ignored. The ADCSSL register specifies the inputs to be scanned. Each bit in the ADCSSL register corresponds to an anlogue input (bit 0 corresponds to AN0, bit 1 to AN1, etc.). If the bit of the corresponding input is set, the selected input is the part of the scanning sequence. The inputs are always scanned from the lower to the higher numbered inputs.

The control register ADCSSL only specifies the input of the positive input of the channel. The CH0NA bit still selects the input of the negative input of the channel during scanning.

If the alternate selection ALTS control bit is set, the scanning only applies to the MUX A (CH0SA<3:0>) input selection. The same applies for the MUX B input selection (CH0SB<3:0>) if selected by the ADCSSL control register.

NOTE: If the number of scanned inputs selected is greater than the number of samples taken per A/D converter interrupt, the higher numbered inputs will not be sampled.

Attention!
The registers ADCHS, ADPCFG, and ADCSSL enable software configuration of the analogue pins AN13 – AN15 not implemented in the microcontroller dsPIC30F4013. If A/D conversion of these inputs is performed, the results is “0”.

Enabling the A/D converter module – When the ADON bit (ADCON1<15>) is set, the module is in active mode and is fully powered and functional. When the ADON bit is cleared, the module is disabled. The digital and analogue portions of the circuit are turned off for maximum current savings.

7.1.3 Starting A/D conversion process – start of the sampling process

Depending on the A/D converter mode selected, the sampling and conversion processes can be started manually or automatically. By starting the conversion process the sampling process on the Sample/Hold amplifier is started.

Manual start of the sampling process – Setting the SAMP bit (ADCON1<1>) causes the A/D converter to begin sampling. One of several options can be used to end sampling and complete the conversiton.Sampling will not resume until the SAMP bit is once again set. Fig. 7-2 shows an example of the manual sample start.

Automatic start of the sampling process – Setting the ASAM bit (ADCON1<2>) causes the A/D converter to automaticaaly begin sampling whenever previous conversion cycle is completed. One of several options can be used to end sampling and complete the conversion. Sampling resumes after the conversion is completed. Fig. 7-3 shows an example of the automatic sample start.

7.1.4 Stopping sampling and starting conversions

The conversion trigger source will terminate sampling and start the sequence of conversions. The SSRC<2:0> control bits (ADCON1<7:5>) select the source of the conversion trigger. Similarly to starting sampling process, a user can start conversions manually or automatically.

Attention!
The SSRC selection bits should not be changed when the A/D module is enabled (ADON=1). If the user wishes to change the conversion trigger source, the A/D module should be diabled first by clearing the ADON bit (ADON=0).

Manual conversion start – When SSRC<2:0>=000, the conversion trigger is under software control. Clearing the SAMP bit (ADCON1<1>) strts the conversion sequence. Fig. 7-2 is an example of the manual start of the conversion sequence.

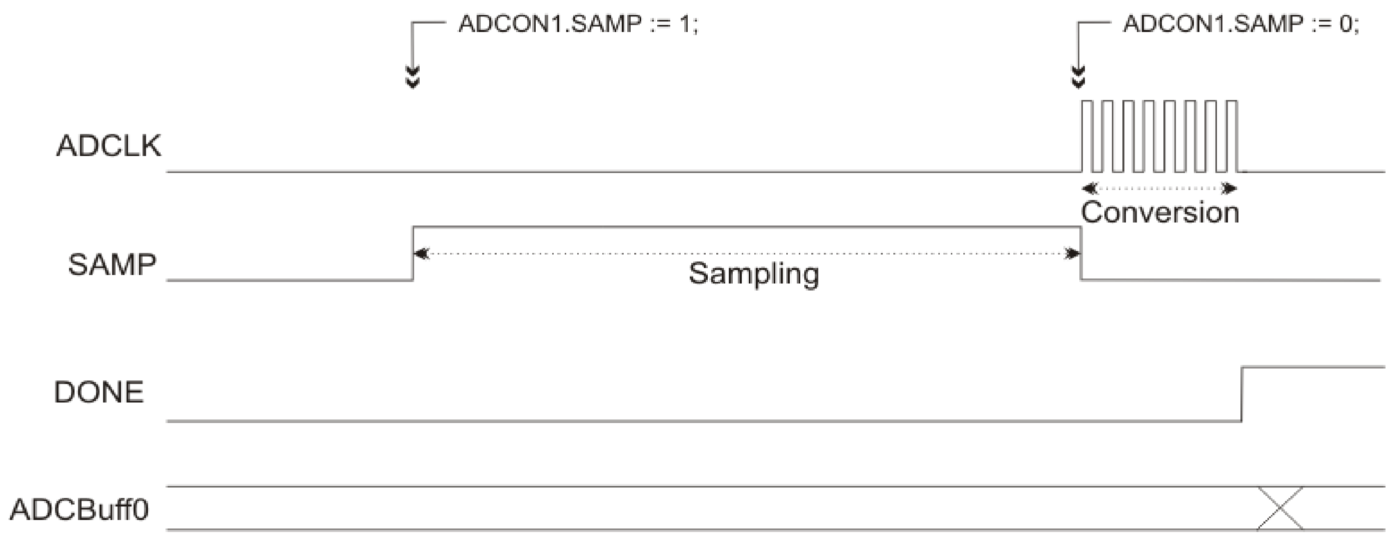


Fig. 7-2 An example of the manual sample start and manual conversion start

The figure shows manual sample start by setting the SAMP bit (SAMP=1) and manual conversion start by clearing the SAMP bit (SAMP=0). The user must time the setting and clearing the SAMP bit in accordance with the acquisition time TSAMP of the input signal.

Example:

This example shows a manual sample start and a manual conversion start. The result of the A/D conversion is sent to the output of port D.

```
/* device = dsPIC30F6014A

   Clock=10MHz */



void main(){

  TRISB  = 0xFFFF;          //Port B is input

  TRISD  = 0;               //Port D is output (for ADC results)

  ADPCFG = 0xFBFF;          //10th channel is sampled and coverted

  ADCON1 = 0;               //ADC off, output_format=INTEGER

                            //Manual start of convesion

                            //Manual start of sampling

  ADCHS  = 0x000A;          //Connect RB10 on AN10 as CH0 input

  ADCSSL = 0;               //No scan

  ADCON3 = 0x1003;          //ADCS=3 (min TAD for 10MHz is 3*TCY=300ns)

  ADCON2 = 0;               //Interrupt upon completion of one
sample/convert

  ADCON1.F15 = 1;           //ADC on

  while (1) {

      ADCON1.F1 = 1;        //Start sampling (SAMP=1)
      Delay_ms(100);        //Wait for 100ms (sampling ...)

      ADCON1.F1 = 0;        //Clear SAMP bit (trigger conversion)

      while (ADCON1.F0 == 0)

         asm nop;           //Wait for DONE bit in ADCON1

      LATD = ADCBUF0;       //Output result on port D

  }

}
```

Fig. 7-3 shows an example of an automatic sample start by setting the control bit ASAM (ASAM=1) and a manual conversion start by clearing the control bit SAMP (SAMP=0).

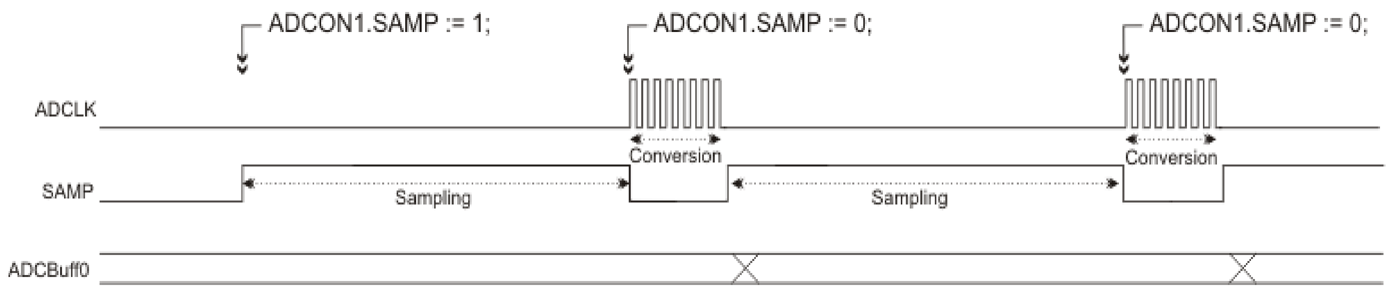


Fig. 7-3 An example of automatic sample start and manual conversion start

Fig. 7-3 shows that by setting the control bit ASAM the automatic sample start is configured; by clearing the control bit SAMP the sampling process is completed and the process of conversion started. Upon completion of the conversion process, A/D converter automatically starts new sampling process by automatically setting the control bit SAMP. When clearing the control bit SAMP the user software has to take into account the minimum acquisition time and the length of the conversion process.

Example:

This example shows an automatic sample start and a manual conversion start. The result of the conversion is sent to the output of port D.

```
/*device = dsPIC30F6014A

  Clock=10MHz*/
```

```
void main(){

  TRISB  = 0xFFFF;              //Port B is input

  TRISD  = 0;                   //Port D is output (for ADC results)

  ADPCFG = 0xFBFF;              //10th channel is sampled and coverted

  ADCON1 = 0x0004;              //ADC off, output_format=INTEGER

                                //Manual start of convesion

                                //Automatic start of sampling after
coversion

  ADCHS  = 0x000A;              //Connect RB10 on AN10 as CH0 input

  ADCSSL = 0;                   //No scan

  ADCON3 = 0x1003;              //ADCS=3 (min TAD for 10MHz is 3*TCY=300ns)

  ADCON2 = 0;                   //Interrupt upon completion of one
sample/convert

  ADCON1.F15 = 1;               //ADC on

  while(1){

      Delay_ms(100);            //Wait for 100ms (sampling ...)

      ADCON1.F1 = 0;            //Clear SAMP bit (trigger conversion)

      while(ADCON1.F1 == 0)

        asm nop;                //Wait for DONE bit in ADCON1

      LATD = ADCBUF0;           //Output result on port D

  }

}
```

7.1.5 External event trigger conversion start

It is often desirable to synchronize the end of sampling and the start of conversion with some external event, i.e. with an external signal source. The A/D module may use one of three sources as a conversion trigger.

External INT pin trigger – When SSRC<2:0>= 001, the A/D conversion is triggered by an active transition on the INT0 pin. The INT0 pin may be programmed for either a rising edge input or a falling edge input.

GP timer compare trigger – This mode is also called trigger mode. By setting the control bits SSRC<2:0>=010, the conversion trigger source can be the general purpose 32-bit timer module 2/3. When the value of the TMR3/TMR2 counter register is equal to the value of the preset (period) register PR3/PR2, a special ADC event signal is generated which triggers the conversion process.

NOTE: The ability of generating an event trigger signal has only timer 3 module. In the 16-bit mode it can initiate the start and also in the 32-bit mode it can do that in the concatenated timer 2/3. This feature does not exist for the timer module 4/5.

Motor control PWM trigger – In the microcontrollers containing this module, like the family dsPIC30F60xx, the start of conversion can be synchronized to the PWM time base. When SSRC<2:0>=011, the A/D sampling and conversion time occur at any user programmable point within the PWM period. In this way the user can minimize the delay between the time when A/D conversion results are acquired and the time when the duty cycle value is updated.

The descibed modes of external initiation or stopping the sampling process or starting the conversion process by setting the control bits SSRC<2:0> = 001, or 010, or 011 could be combined with the automatic sample start by setting the ASAM bit. Figs. 7-6 and 7-7 give examples of initiating the convesrion process with the manual and automatic sample start, respectively.

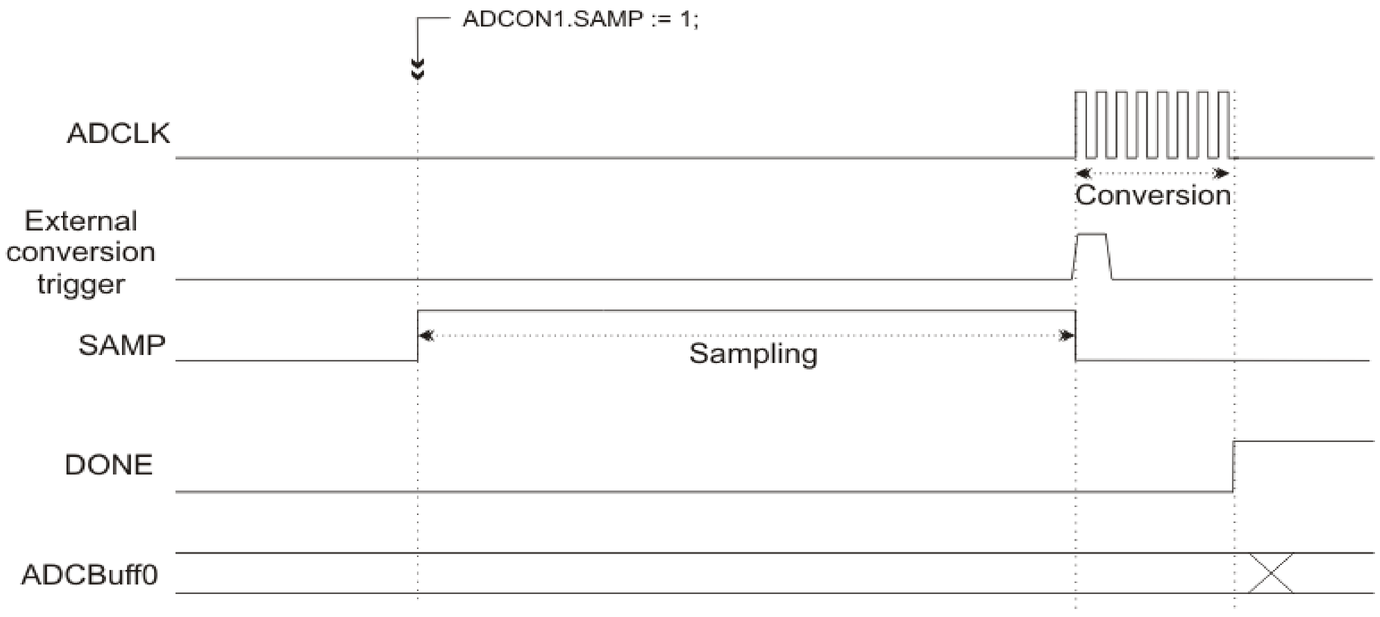


Fig. 7-6 External initiation of the conversion and manual sample start.

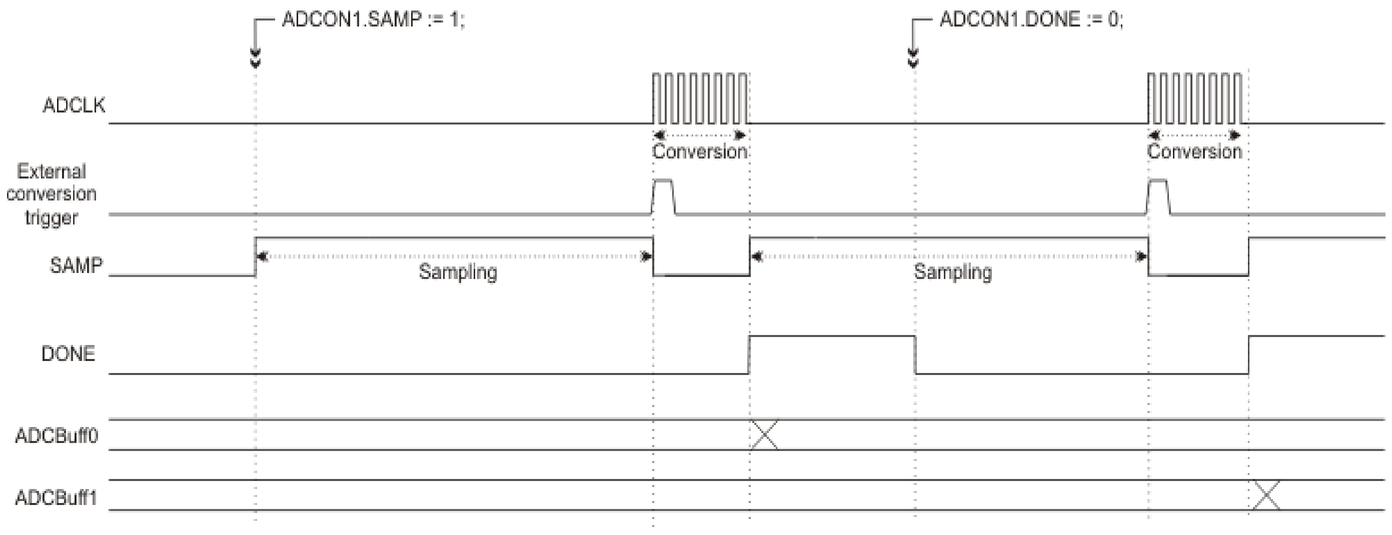


Fig. 7-7 External initiation of the conversion and automatic sample start.

NOTE: Depending on the start modes of the sampling and conversion, one obtains different sampling times. In all cases, however, it is required that sampling time TSAMP is longer than the minimum time determined by device charcteristics.

Example:

This example shows automatic sample start and start of conversion by timer 3. The result of the conversion is sent to the output of port D.

```
/*device = dsPIC30F6014A

  Clock=10MHz*/



void main(){

  TRISB   = 0xFFFF;        //Port B is input

  TRISD   = 0;             //Port D is output (for ADC results)

  ADPCFG = 0xFBFF;         //10th channel is sampled and coverted

  ADCON1 = 0x0040;         //ADC off, output_format=INTEGER

                           //Timer 3 starts convesion
  ADCHS   = 0x000A;         //Connect RB10 on AN10 as CH0 input

  ADCSSL = 0;               //No scan

  ADCON3 = 0x0F00;          //TAD = internalTCY

                            //Sample time ends with timer 3 clock

  /*TIMER3 init*/

  TMR3    = 0;              //Reset TIMER3 counter

  PR3     = 0x3FFF;         //PR3, number of TIMER3 clocks between two 
conversions start

  T3CON   = 0x8010;         //TIMER3 ON, prescale 1:1



  ADCON1.F15 = 1;           //ADC on

  ADCON1.F2  = 1;           //ASAM=1, start sampling after conversion ends

  while(1){

     ADCON1.F1 = 1;         //SAMP=1, start sampling

     while(ADCON1.F0 == 0)

       asm nop;             //Wait for DONE bit in ADCON1

     LATD = ADCBUF0;        //Output result on port D

  }

}
```

7.1.6 Controlling sample/conversion operation

The application software may poll the SAMP and CONV bits to keep track of the A/D conversions. Alternatively, the module can generate an interrupt (ADIF) when conversions are complete. The application software may also abort A/D operations if necessary.

Monitoring sample/conversion status – The SAMP (ADCON1<1>) and CONV (ADCON1<0>) bits idicate the sampling state and the conversion state of the A/D, respectively. Generally, when the SAMP bit clears indicating end of sampling, the CONV bit is automatically set indicating start of conversion. Clearing of the CONV bit denotes end of conversion.. If both SAMP and CONV bits are 0, the A/D is in an inactive state. In some operational modes, the SAMP bit may also invoke and terminate sampling and the CONV bit may terminate conversion.

Generating an A/D interrupt - The SMPI<3:0> bits control the generation of interrupts. The interrupt will occur after the number of sample/conversion sequences specified by the SMPI bits and re-occur on each equivalent number of samples. The vaule specified by the SMPI bits will correspond to the number of data samples in the RAM buffer, up to the maximum of 16. Disabling the A/D interrupt is not done with the SMPI bits. To disable the interrupt, clear the ADIE (IEC0<11>) analogue module interrupt enable bit.

Aborting sampling – Clearing the SAMP bit while in manual sampling mode will terminate sampling, but may also start a conversion if SSRC= 000. Clearing the ASAM bit while in automatic sampling mode will not terminatean on going sample/convert sequence, however, sampling will not automatically resume after a subsequent conversion.

Aborting a conversion – Clearing the ADON bit during a conversion will abort the current conversion. The A/D result will not be updated with the partially completed A/D conversion, i.e. the corresponding ADCBUFi buffer will contain the value od the last completed conversion (or the last value written to the buffer).

7.1.7 Writing conversion results into the buffer

As conversions are completed, the module writes the results of the conversions into the RAM buffer. This buffer is a RAM array of sixteen 12-bit words. The buffer is accessed through the 16 address locations within the SFR space, named ADBUF0, ADBUF1... ADBUFF. User software may attempt to read each A/D result as it is generated, but this might consume too much CPU time. Generally, the module will fill the buffer with several A/D conversion results and then generate an interrupt. This allows accomplishment of high sampling rates.

Number of conversions per interrupt – The SMPI<3:0> bits (ADCON2<5:2>) will select how many A/D conversions will take place before the CPU is interrupted. This number can vary from 1 to 16 samples per interrupt. The conversion results will always be written to ADCBUF0, ADCBUF1... ADCBUFF. If SMPI<3:0>= 0000, an interrupt is generated after each conversion and the result is always written in ADCBUF0.

Attention!!!
When the BUFM bit (ADCON2<1>) is set, the user can not program the SMPI bits to a value that specifies more than 8 conversions per interrupt.

Buffer fill mode – In this mode two 8-word groups of the RAM buffer memory will alternately receive the conversion results after each interrupt. This mode is selected by setting the control bit BUFM (ADCON2<1>). While one group receives the A/D conversion results, the content of the other group may be moved to the data memory. In this way the CPU time i saved and high sampling rates can be accomplished. Upon initialization of this mode, by setting the value of the SMPI bit to n (n = 8), the result of each A/D conversion will be loaded to the lower group of the RAM buffer memory, i.e. to the locations ADBUF0 to ADBUF7. After n A/D conversions are completed, an A/D converter interrupt generated, and the interrupt routine carried out, the higher group of the RAM buffer memory, ADBUF8 to ADBUFF, is selected and the results of the next n A/D conversions (n = 8) are loaded to these locations. This mode is selected when the processor cannot unload the RAM buffer within one sample and conversion time sequence.

NOTE: In the Buffer fill mode when the BUFM bit is set, the BUFS status bit (ADCON2<7>) indicates the half of the buffer that the A/D converter is currently filling. If BUFS=0, then the A/D converter is filling the lower group, ADCBUF0 – ADCBUF7, and the user software could read conversion values from the higher group, ADCBUF8 – ADCBUFF. If BUFS=1, then the A/D converter is filling the higher group, ADCBUF8 – ADCBUFF, and the user software could read conversion values from the lower group, ADCBUF0 – ADCBUF7.

7.1.8 A/D sampling requirements

For proper functioning of the A/D converor the total sampling time should take into acount the sample/hold amplifier settling time, holding capacitor charge time, and temperature.

The analogue input model of the A/D converter is shown in Fig.7-8. For the A/D converter to meet its specified accuracy, the charge holding capacitor (CHOLD) must be allowed to fully charge to the voltage level on the analogue input pin (VA). The source impedance (RS) and the internal sampling switch impedance (RSS) directly affect the time required to charge the capacitor CHOLD. Furthermore, the sampling switch impedance (RSS) varies with the device supply voltage VDD. All these influences, however small, affect the total sampling time of the A/D converter. To avoid exceeding the limit of the sampling time and to minimize the effects of pin leakage currents on the accuracy of the A/D converter, the maximum recommended source impedance (RS) is 2.5kO.

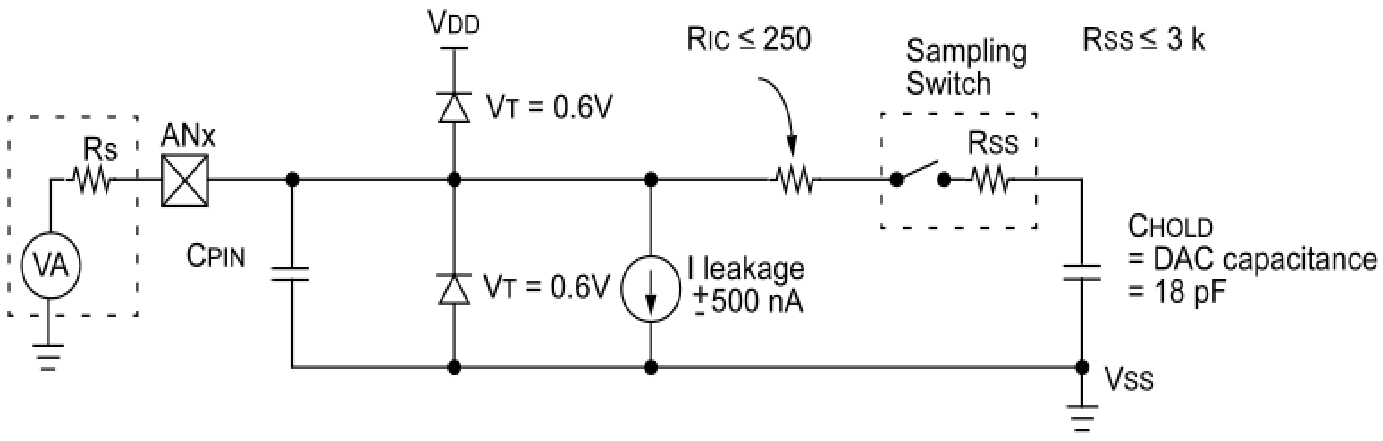


Fig. 7-8 The analogue input model of the A/D converter.

To calculate the sampling capacitor charging time the following equation may be used:

TSAMP = TAMP + TC +TCOEF ,

where TSAMP is the total sampling time, TAMP is the sample/hold amplifier settling time, TC is the holding capacitor charging time, and TCOEF is the time due to temperature variation.

NOTE: Sample/hold amplifier settling time is 0.5µs.

A/D holding capacitor charging time is given by:

TC = -CHOLD(RIC + RSS + RS)ln(1/2n) in seconds,

where n is the number of levels of the A/D converter (for a 12-bit A/D converter this is 4096). The time due to temperture dependence of the A/D converter is given by:

TCOEF = (Temp – 25C)(0.005µs/C).

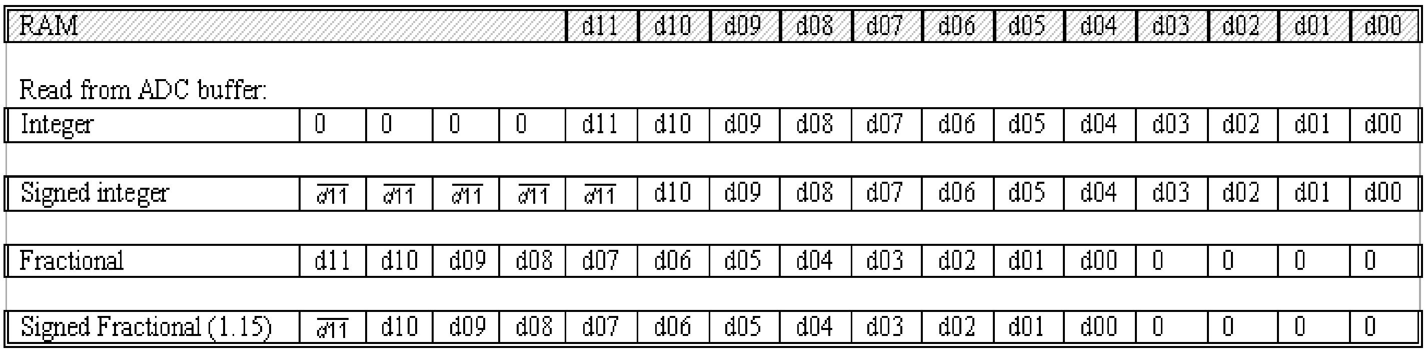
Example:

For the values: CHOLD=18pF, RIC = 250O, minimum RS =1O, n=4096 for the full range of a 12-bit A/D converter, VDD=5V, RSS =1.2kO, and operating temperature 25C the following times are obtained TC =0.24µs and TSAMP =0.74 µs.

For the values: CHOLD=18pF, RIC = 250O, maximum RS =2.5kO, n=4096 for the full range of a 12-bit A/D converter, VDD=5V, RSS =1.2kO, and operating temperature 25C the following times are obtained TC =0.641 µs and TSAMP =1.14 µs.

7.1.9 Reading the A/D result buffer

The results of A/D conversions are loaded into the 16-word 12-bits wide RAM buffer. Reading the RAM buffer is possible from the special function register memory locations ADBUF0 to ADBUFF in one of the four selectable formats: integer, signed integer, fractional, and signed fractional. The FORM<1:0> bits (ADCON1<9:8>) select the format. Fig. 7-9 shows the data output formats of the A/D converter.



RAM					d11	d10	d09	d08	d07	d06	d05	d04	d03	d02	d01	d00
Read from ADC buffer:																
Integer	0	0	0	0	d11	d10	d09	d08	d07	d06	d05	d04	d03	d02	d01	d00
Signed integer	$\overline{d11}$	$\overline{d11}$	$\overline{d11}$	$\overline{d11}$	$\overline{d11}$	d10	d09	d08	d07	d06	d05	d04	d03	d02	d01	d00
Fractional	d11	d10	d09	d08	d07	d06	d05	d04	d03	d02	d01	d00	0	0	0	0
Signed Fractional (1.15)	$\overline{d11}$	d10	d09	d08	d07	d06	d05	d04	d03	d02	d01	d00	0	0	0	0

Fig. 7-9 Data output formats of the A/D converter.

Attention!!!
The analogue inputs of the A/D converter are diode protected against overvoltages. In case that the voltage at an anlogue input is lower by 0.3V than VSS or higher by 0.3V than VDD the corresponding diode will become forward biased. If there is no current limiting at the input, the device may be damaged.

NOTE: An external RC filter is sometimes added to the analogue inputs to limit the frequency of input signals. The R component should be selected to ensure that the value of 2.5kO is not exceeded, i.e. the sampling time requirements are satisfied.

7.1.10 Operation during SLEEP and IDLE modes

During SLEEP or IDLE modes, depending upon the configuration, the operation of the A/D converter is possible. These modes are useful for minimizing conversion noise because the digital activity of the CPU, buses, and other peripherals is minimized.

Operation in SLEEP mode without RC A/D clock – If the internal RC clock generator is not used, when the device enters SLEEP mode, all clock sources to the module are shut down. If SLEEP occurs in the middle of a conversion, the conversion is aborted. The converter will not resume a partially completed conversion on exiting from SLEEP mode. Register contents are not affected by the device entering or leaving SLEEP mode.

Operation in SLEEP mode with RC A/D clock – If the A/D clock source is set to the internal RC oscillator (ADRC=1), the A/D module can operate during SLEEP mode. This eliminates digital switching noise. When the conversion is completed, the CONV bit will be cleared and the result loaded into the RAM buffer.

If the interrupt is enabled ADIE=1 (IEC0<11>), the device will wake-up from SLEEP when the A/D interrupt occurs ADIF=1 (IPS0<11>). The interrupt service routine will be carried out if the A/D'interrupt is greater than the current CPU priority. Otherwise, the device returns to SLEEP mode.

If the A/D interrupt is not enabled, the A/D module will then be turned off while the ADON bit remains set.

To minimize the effects of digital noise on the A/D module operation, the user shoud select a convesrion trigger source before the microcontroller enters SLEEP mode. The control bits SSRC<2:0>=111 are set befor the PWRSAV instruction.

Attention!!!
If the operation of the A/D converter during SLEEP mode is required, the user has to select the internal RC clock generator (ADRC=1)

Operation during IDLE mode – If the control bit ADIDL (ADCON1<13>) is cleared, the A/D module will continue normal operation in IDLE mode. If the A/D interrupt is enabled (ADIE=1), the device will wake-up from IDLE mode when the A/D interrupt occurs. The A/D interrupt service routine will be carried out if the A/D interrupt is greater than the current CPU priority. Otherwise, the device returns to IDLE mode.

If A/D interrupt is disabled, the A/D module will be turned off while the ADON bit remains set.

If the control bit ADIDL is set, the module will stop in IDLE mode. If the device enters IDLE mode in the middle of a conversion, the conversion is aborted. The converter will not resume a partially completed conversion on exiting from IDLE mode.

Attention!!!
A device RESET forces all registers to their RESET state. This forces the A/D module to be turned off and any conversion in prgress is aborted. All pins will be configured as analogue inputs. The RAM buffer will contain unknown data.

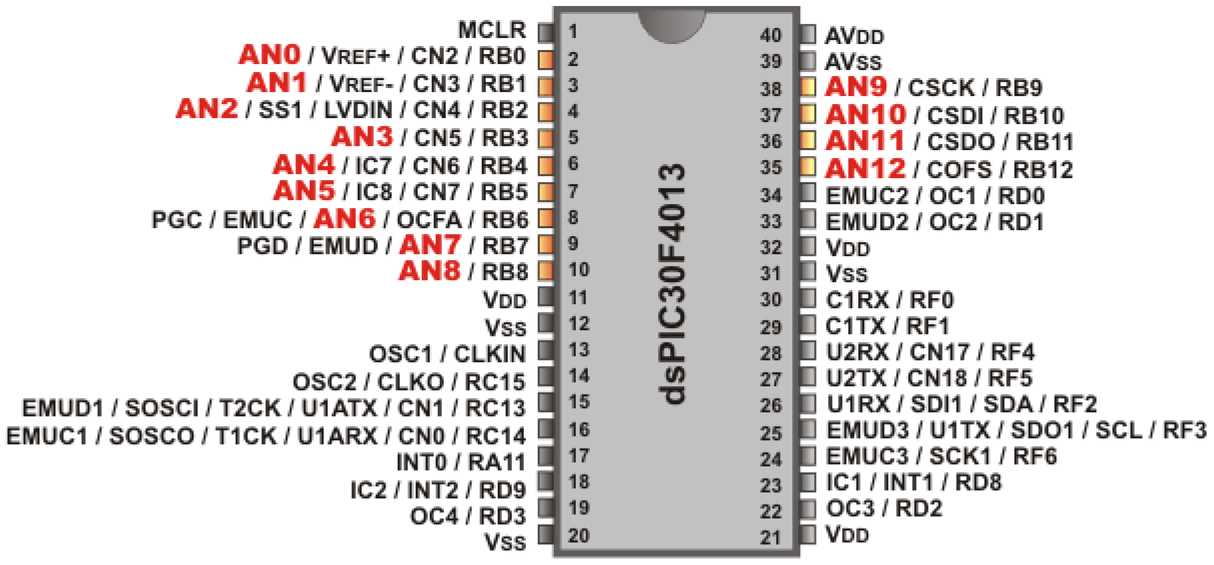


Fig. 7-10a Pinout of dsPIC30F4013.

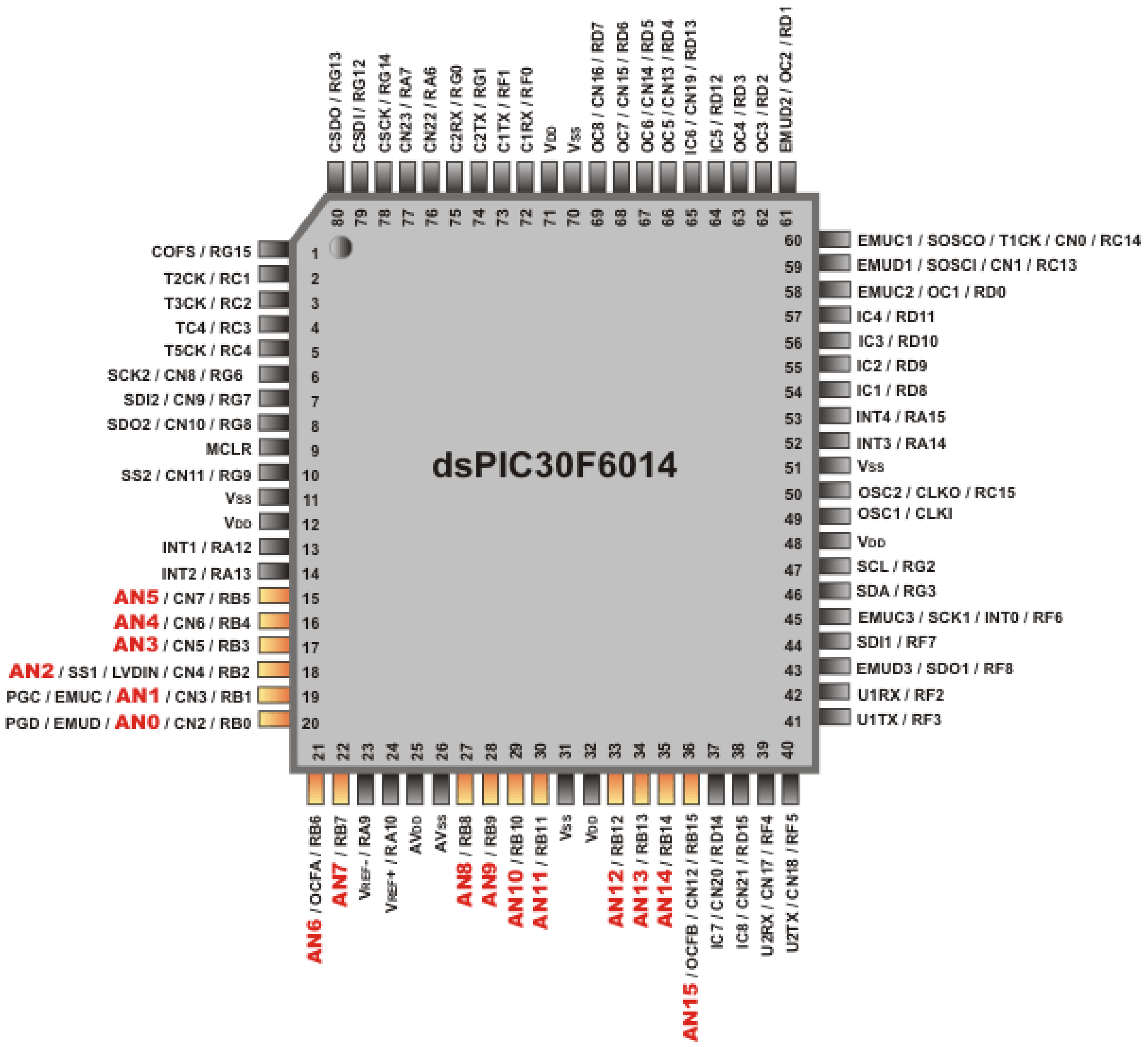


Fig. 7-10b Pinout of dsPIC30F6014A.

Finally, the desriptions of the A/D module control registers are presented.

name	ADR	15-12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
ADBUF0	0x0208	-	ADC DATA BUFFER 0												0x0uuu

ADBUF1	0x0208	-	ADC DATA BUFFER 1	0x0uuu
ADBUF2	0x0208	-	ADC DATA BUFFER 2	0x0uuu
ADBUF3	0x0208	-	ADC DATA BUFFER 3	0x0uuu
ADBUF4	0x0208	-	ADC DATA BUFFER 4	0x0uuu
ADBUF5	0x0208	-	ADC DATA BUFFER 5	0x0uuu
ADBUF6	0x0208	-	ADC DATA BUFFER 6	0x0uuu
ADBUF7	0x0208	-	ADC DATA BUFFER 7	0x0uuu
ADBUF8	0x0208	-	ADC DATA BUFFER 8	0x0uuu
ADBUF9	0x0208	-	ADC DATA BUFFER 9	0x0uuu
ADBUFA	0x0208	-	ADC DATA BUFFER 10	0x0uuu
ADBUFB	0x0208	-	ADC DATA BUFFER 11	0x0uuu
ADBUFC	0x0208	-	ADC DATA BUFFER 12	0x0uuu
ADBUFD	0x0208	-	ADC DATA BUFFER 13	0x0uuu
ADBUFE	0x0208	-	ADC DATA BUFFER 14	0x0uuu
ADBUFF	0x0208	-	ADC DATA BUFFER 15	0x0uuu

Table 7-2 Description of the RAM buffer registers.

NOTE: Unimplemented bits read "0".

name	**ADR**	**15**	**14**	**13**	**12-10**	**9**	**8**	**7-5**	**4-3**	**2**	**1**	**0**	**Reset State**
ADCON1	0x02A0	ADON	-	ADIDL	-	FORM<1:0>		SSRC<2:0>		ASAM	SAMP	CONV	0x0000

Table 7-3 Description of the configuration register ADCON1

```
ADON – A/D operating mode bit
       (ADON=1 A/D converter module is operating, ADON=0  A/D converter is
off)
ADIDL – Stop in IDLE mode bit (ADIDL=1 discontinue module operation when
device enters
        IDLE mode, ADIDL=0 continue module operation in IDLE mode)
FORM<1:0> - Data output format bits
     00 – integer
     01 – signed integer
     10 – fractional
     11 – signed fractional
SSRC<2:0> - Conversion trigger source select bit
     000 – clearing SAMP bit ends sampling and starts conversion
     001 – active transition on INTO pin ends sampling and starts
conversion
     010 – general purpose timer3 compare ends sampling and starts
conversion
     011 – motor control PWM interval ends sampling and starts conversion
     101...110 – reserved
     111 – automatic mode
ASAM – A/D sample auto-START bit
SAMP – A/D sample enable bit (SAMP=1 at least one A/D sample/hold amplifier
sampling,
       SAMP=0 sample/hold amplifiers are holding)
DONE – A/D conversion status bit
       (DONE=1 A/D conversion is done, DONE=0 A/D conversion is in
progress)
```

NOTE: Unimplemented bits read "0".

name	ADR	15-13	12-11	10	9-8	7	6	5	4	3	2	1	0	Reset State
ADCON2	0x02A2	VCFG<2:0>	-	CSCNA	-	BUFS	-	SMPI<3:0>				BUFM	ALTS	0x0000

Table 7-4 Description of the configuration register ADCON2

```
VCFG<2:0> - Voltage reference configuration bit
     000 - VREFH =AVDD,  VREFL=AVSS
     001 - VREFH= External VREF+ pin, VREFL=AVSS
     010 -  VREFH =AVDD,  VREFL=External VREF- pin
     011 - VREFH= External VREF+ pin, VREFL=External VREF- pin
     1xx - VREFH =AVDD,  VREFL=AVSS
CSCNA - Scan input selections for CH0+ S/H input for MUX A input
multiplexer setting bit
BUFS - Buffer full status bit
    Only valid when BUFM=1 (ADRES split into 2 x 8-word buffers)
    BUFS=1 A/D is currently filling higher buffer 0x8-0xF,
    BUFS=0 A/D is currently filling lower buffer 0x0-0x7
SMPI<3:0> - Sample/convert sequences per interrupt selection bits
     0000 - interrupts at the completion of conversion for each
sample/convert sequence
     0001 - interrupts at the completion of conversion for each 2nd
sample/convert sequence
     ...
     1110 - interrupts at the completion of conversion for each 15th
sample/convert sequence
     1111 - interrupts at the completion of conversion for each 16th
sample/convert sequence
BUFM - Buffer mode select bit (BUFM=1 Buffer configured as two 8-word
buffers
       ADCBUF(15...8), ADCBUF(7...0),
       BUFM=0 Buffer configured as one 16-word buffer ADCBUF(15...0))
ALTS - Alternate input sample mode select bit
       (ALTS=1 alternate sampling enabled, ALTS=0 always use MUX A input)
```

NOTE: Unimplemented bits read "0".

name	ADR	15-13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
ADCON3	0x02A4	-	SAMC<4:0>					ADRC	-	ADCS<5:0>						0x0000

Table 7-5 Description of the configuration register ADCON3

```
SAMC<4:0> - Auto sample time bits
     00000 - 0 TAD
     00001 - 1 TAD
     ...
     11111 - 31 TAD
ADRC - A/D conversion clock source bit (ADRC=1 A/D internal RC clock,
       ADRC=0 Clock derived  from system clock)
ADCS<5:0> - A/D conversion clock select bits
     000000 - TCY/2 * (ADCS<5:0> + 1)=TCY/2
     000001 - TCY/2 * (ADCS<5:0> + 1)=TCY
     ...
     111111 - TCY/2 * (ADCS<5:0> + 1)=32*TCY
```

NOTE: Unimplemented bits read "0".

name	ADR	15-13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
ADCHS	0x02A6	-	CH0NB	CH0SB<3:0>				-	-	-	CH0NA	CH0SA				0x0000

Table 7-6 Description of the ADCHS configuration register

```
CH0NB - Channel 0 negative input select for MUX B multiplexer setting bit
        (CH0NB=1 input AN1 selected, CH0NB=0 input VREF- selected)
CH0SB<3:0> - Channel 0 positive input select for MUX B multiplexer setting
bit
      0000 - input AN0 selected
      0001 - input AN1 selected
      ...
      1110 - input AN14 selected
      1111 - input AN15 selected
CH0NA - Channel 0 negative input select for MUX A multiplexer setting bit
        (CH0NA=1 input AN1 selected, CH0NA=0 input VREF- selected)
CH0SA<3:0> - Channel 0 positive input select for MUX A multiplexer setting
bit
      0000 - input AN0 selected
      0001 - input AN1 selected
      ...
      1110 - input AN14 selected
      1111 - input AN15 selected
```

name	ADR	15-0	Reset state
ADPCFG	0x02A8	PCFG<15:0>	0x0000

Table 7-7 Description of the ADPCFG configuration register

```
PCFG<15:0> - Analogue input pin configuration control bits
(PCFG(i)=1 pin ANi configured as digital I/O pin,
PCFG(i)=0 pin ANi configured as analogue input pin)
```

name	ADR	15-0	Reset state
ADCSSL	0x02AA	ADC Input scan select register	0x0000

Table 7-8 Description of the ADCSSL configuration register

```
ADCSSL<15:0> - A/D input pin scan selection bits
```

Chapter8: Memory Model

Introduction

The organization of memory and the mode of its use are called the memory model. Understanding the memory model is essential for programming. The memory is divided into the program memory and data memory. The program memory is further divided into the user program space and the user configuration space. The data memory is the space of memory locations used by a program for saving different data required for the program execution. The size of the memory space is different for different members of the dsPIC30F family.

8.1 Program memory

The dsPIC30F microcontrollers have 4M 24-bit (3 bytes) program memory address space. This does not mean that the the size of the memory is 12MB (3x4M=12MB), but that is can generate 4M(4x220) different addresses.

The size of the program memory of a dsPIC30F4013 device is 16K words, i.e. 3x16K=48KB. A program can have 16K instructions less the number of auxiliary locations (interrupt table and similar).

The structure of the program memory is given in Fig.8.1 The first two locations are reserved for defining the beginning of a program. The execution of a program starts from

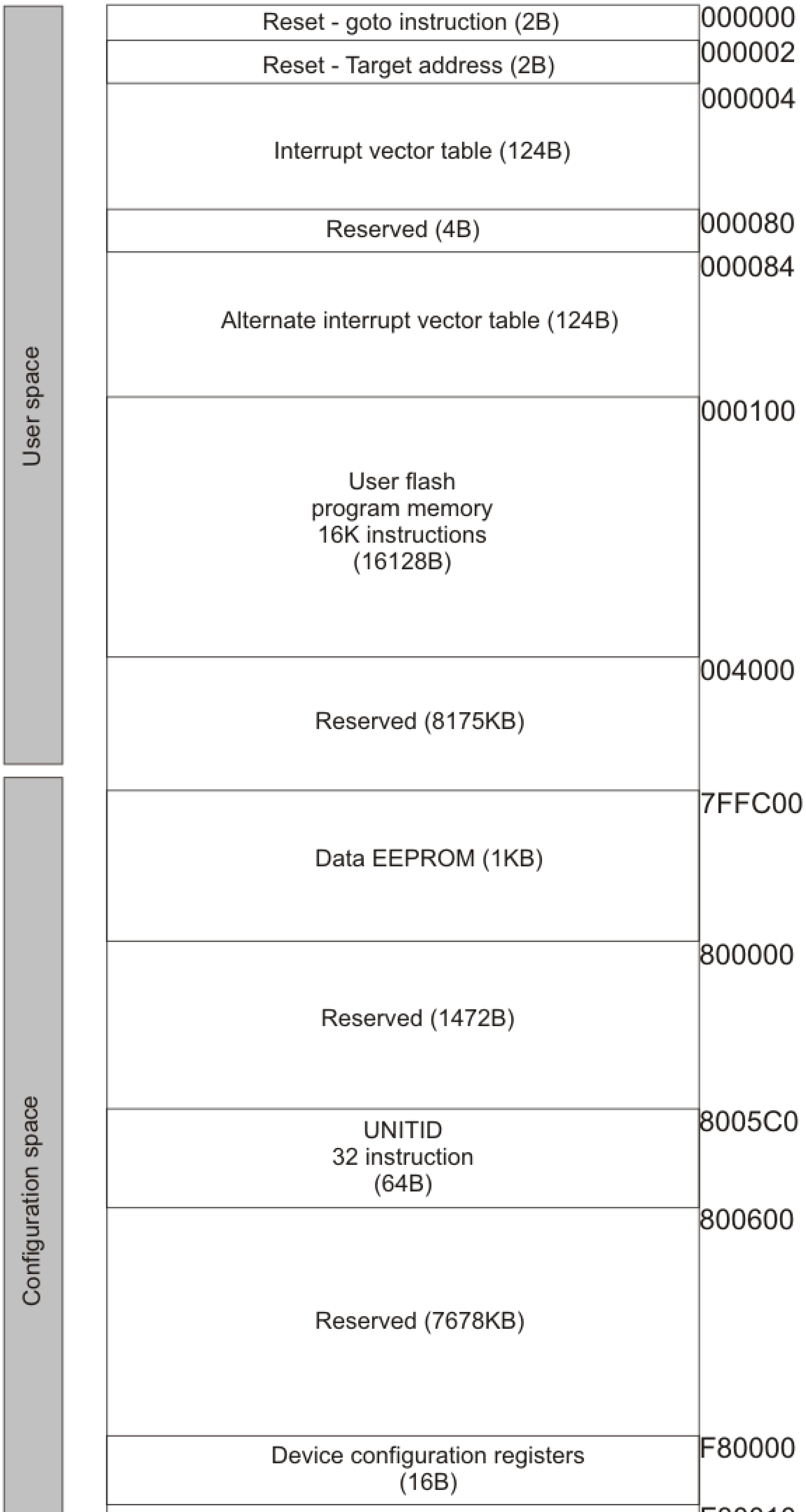

User space
Configuration space
Reset - goto instruction (2B)
Reset - Target address (2B)
Interrupt vector table (124B)
Reserved (4B)
Alternate interrupt vector table (124B)
User flash
program memory
16K instructions
(16128B)
Reserved (8175KB)
Data EEPROM (1KB)
Reserved (1472B)
UNITID
32 instruction
(64B)
Reserved (7678KB)
Device configuration registers
(16B)
000000
000002
000004
000080
000084
000100
004000
7FFC00
800000
8005C0
800600
F80000

Fig. 8.1 Program memory of dsPIC30F4013 microcontroller

The program memory is accessible only via even addresses. An attempt to read an odd address will result in CPU trap and device reset. Interrupts and traps are described in Chapter 3.

Example:

The main program start at the location 0x000120. The location 0x000000 contains GOTO instruction, and location 0x000002 contains value 0x000120. During each reset of the device, from the location 0x000000 GOTO instruction is read and from the location 0x000002 the address is read where to jump (in this case 0x000120). The program starts execution from the location 0x000120.

After the first two locations is the interrupt vector table serving, as explained in Chapter 3, for specifying the locations where the interrupt subprograms are. It consists of 62 locations (see Chapter 3). The next two locations (0x000080 and 0x000082) are reserved for internal purposes. Then, there is the altenative interrupt vector table with another 62 locations. At the address 0x000100 is the start of the space in the program memory for the main program and subprograms. The maximum address of the program space is 0x0007FFE (total of 16K locations). After this address is the program memory configuration space.

8.2 Data memory (RAM)

Data memory (RAM)serves for storing and keeping data required for the proper operation of the programs. Depending on the program in progress, it can, but does not have to, be split into two sections. For DSP2 instruction set the data memory is considered to consist of two sectioins. The two data spaces are accessed by using two address generation units and separate data paths, i.e. two data can be read or written simultaneously. Chapter 11 gives more details. For other instructions the data memory is considered unique. Structure of the data memory of the dsPIC30F4013 and dsPIC30F6014A devices is shown in Fig. 8-2.

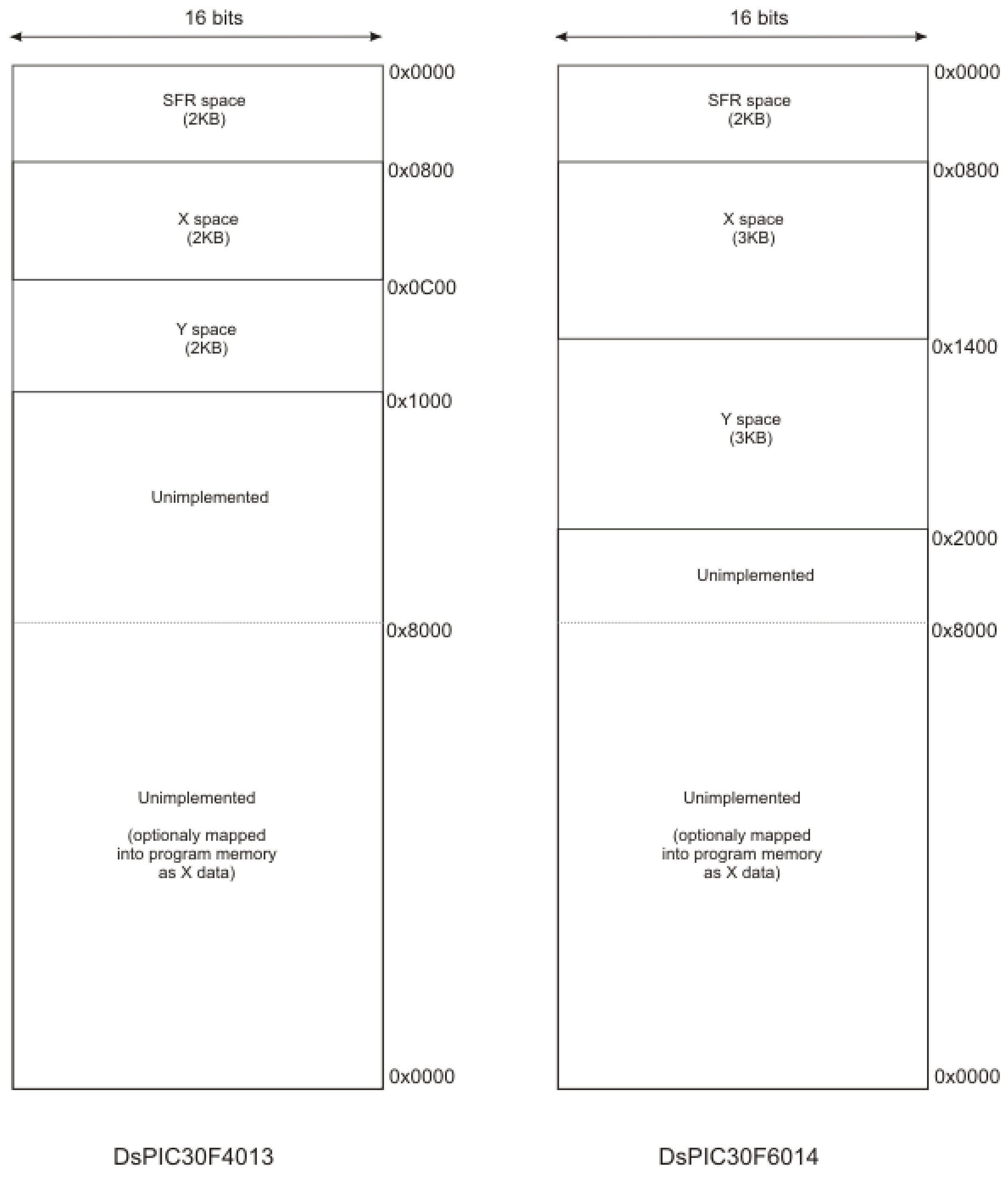


Fig. 8-2 Data memory of dsPIC30F4013

The size of the data memory, similarly to the program memory, depends on the model of dsPIC devices. For dsPIC30F4013 device the data memory has 64K. The addresses are 16-bit, i.e. no more than 64K addresses can be generated. All data memory addresses are even. An attempt to access an odd address will result in device reset. The first 2K locations are reserved for the Special Function Registers (SFR) and general purposes (1K 16-bit locations). These registers contain the control and status bits of the device. From the address 0x0800 the RAM is divided into two sections, X and Y. This division is essential only for DSP instructions. For other applications only the total size matters (e.g. for dsPIC30F4013 this size is 2KB). Writing to the RAM is performed as if this division did not exist for both DSP instructions and others. Only when reading DSP instructions this division is used to read two memory locations simultaneously. This access to the memory speeds up considerably the execution of DSP instructions which have been optimized for signal processing, which most of the time requires calculation of the sums of products of two arrays (the so called MAC istructiuons - Multiply and Add to Accumulator). This requires fast reading of both operands to be multiplied and the result added to the previous value (accumulation).

Modulo addressing

Modulo, or circular addressing provides an automated means to support circular data buffers using hardware. The objective is to remove the need that the software has to perform data address boundary checks when executing tightly looped code as is typical in many DSP algorithms. Any W register, except W15 (used as the stack pointer), can be selected as the pointer to the modulo buffer. Also the use of W14 register as the stack frame pointer register is recommended.

A frame is a user defined section of memory in the stack that is used by a single subroutine (function). The length of a circular buffer used in modulo addressing is not directly specified. The maximum possible length is 32K words. A location in the buffer can be either 16-bit (16-bit buffer) or 8-bit (8-bit buffer). However, modulo addressing always operates with 16-bit buffers, therefore the length of the 8-bit buffers **has to be** even. Another **restriction on the 8-bit buffers is that they can not be in the Y space, but only in the X space or in the program memory**.

Modulo addressing is specified by the five registers: XMODSRT, XMODEND, YMODSRT, YMODEND, and MODCON. The register XMODSRT specifies the starting and XMODEND ending address in a circular buffer in the X space, if used. The register YMODSRT specifies the starting and YMODEND ending address in a circular buffer in the Y space, if used. The register MODCON is used for enabling/disabling modulo addressing and for its configuration. The structure of the MODCON register is shown at the end of this chapter. The registers XMODEND and YMODEND contain the address of the last occupied location in the buffer. The address of the last occupied location is always odd in both 16- bit and 8-bit buffers.

Modulo addressing example

```
/*dsPIC30F6014A*/



int buff[10];  //Circular buffer

int i;



void main(){

  TRISD = 0;

  for(i = 0;i <= 9;i++)     //Init buff

    buff[i] = i;
  YMODSRT = &buff;           //YMODSRT points to the first element of buff

  YMODEND = &buff+19;        //YMODEND points to the end address of buff

  MODCON = 0x80AA;           //Moduo address Y space

  asm nop;                   //After changing MODCON forced nop is
recommended



  W10 = &buff;               //W10 points to the first element of buff

  while(1){

     asm{

       MOV [W10++], W5       //Copy current element to W5,prepare for next

       MOV #$02D6, W4        //W4 points to LATD

       MOV W5, [W4]          //Fill LATD with item

     }

     Delay_ms(200);

  }

}
```

Modulo addressing hardware is enabled only with indirect addressing ([W10++]). After the registers for configuring modulo addressing have been set, indirect addressing must not be used. Since the next instruction can be compiled by using indirect addressing, it is recommended that after setting these registers NOP (non-operation) is added. This ensures that the next instruction is executed correctly.

8.4 Bit-reversed addressing

Bit-reversed addressing is used for simplifying and speeding-up the access to the arrays in FFT (Fast Fourier Transform) algorithms. Like in modulo addressing, this part of hardware allows that the part of the program which calculates the elements of an array be skipped in order to simplfy the program and speed-up its execution. This addressing method is possible only in the X space and for data writes only. It is used with the pre-increment or post-increment addressing modes. Modulo addressing and bit-reversed addressing can be enabled simultaneously using the same register. Bit-reversed addressing operation will always take precedence for data writes and modulo addressing for data reads. Therefore, modulo addressing restrictions will aplly when reading data and bit-reversed addressing restrictions when writing data.

Bit-reversed addressing enable and the size of the bit-reversed data buffer are specified by the register XBREV. Bit-reversed addressing is assigned to one of the W registers specified by the MODCON register.

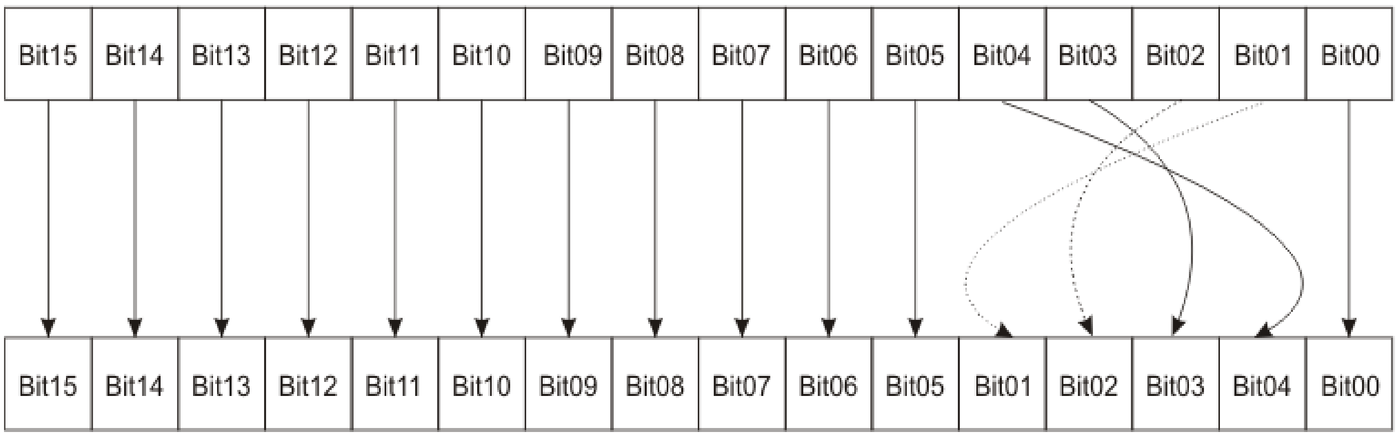


Fig. 8-3 Block diagram of bit-reversed addressing of a 16-word array

Fig. 8-3 shows block diagram of bit-reversed addressing of a 16-word array. The last bit is always zero because the addresses have to be even (access to 16-bit data). The sequence of bit-revessed addresses is given in Table 8-1.

Normal Address					**Bit-reversed**				
R1	R2	R3	R4	decimal	R1	R2	R3	R4	decimal
0	0	0	0	00	0	0	0	0	00
0	0	0	1	01	1	0	0	0	08
0	0	1	0	02	0	1	0	0	04
0	0	1	1	03	1	1	0	0	12
0	1	0	0	04	0	0	1	0	02
0	1	0	1	05	1	0	1	0	10
0	1	1	0	06	0	1	1	0	06
0	1	1	1	07	1	1	1	0	14
1	0	0	0	08	0	0	0	1	01
1	0	0	1	09	1	0	0	1	09
1	0	1	0	10	0	1	0	1	05
1	0	1	1	11	1	1	0	1	13
1	1	0	0	12	0	0	1	1	03
1	1	0	1	13	1	0	1	1	11
1	1	1	0	14	0	1	1	1	07
1	1	1	1	15	1	1	1	1	15

Table 8-1 Bit-reversed address sequence (16-entry)

Table 8-2 shows the dependence of the value loaded into the XBREV register on the size of the bit-reversed data buffer.

Buffer Size (16-bit word)	XBREV (XB<14:0>)
1024	0x0200
512	0x0100
256	0x0080
128	0x0040
64	0x0020
32	0x0010
16	0x0008
8	0x0004
4	0x0002
2	0x0001

Table 8-2 Bit-revesrsed address modifier values

By using bit-reversed addressing when calculating FFT, the process is several times faster. For the dsPIC30F4013 device the size of the bit-reversed data buffer is limited to 1K words.

Example of bit-reversed addressing:

```
/*dsPIC30F6014A*/



int InBuff[16];          //input buffer

int BrBuff[16];          //bit-reversed buffer

int i;



void main(){

  TRISD = 0;             //PortD is output port

  TRISB = 0;

  for(i =0; i<=15; i++){

      InBuff[i] = i;     //Init input buffer

      BrBuff[i] = 0;     //clear bit-rev buffer
  }



  XBREV = 0x8008;        //enable bit-reversed addressing for 16-word
buffer

  MODCON = 0x01FF;       //setup MODCON for W1 bit-rev register

  asm nop;               //after changing MODCON forced nop is recomended



  W0 = &InBuff;          //W0 points to the first element of input buffer
(InBuff)

  W1 = &BrBuff;          //W1 points to the first element of output buffer
(BrBuff)



  asm{

    repeat #15

    mov [W0++], [W1++]   //fill output buffer (BrBuff)

  }



  XBREV = 0;             //disable bit-reversed addressing

  asm nop;



  while(1){              //SHOW bit-rev buffer on PORTD

   for(i =0; i<=15; i++){

     LATB = i;

     LATD = BrBuff[i];
     Delay_ms(2000);


   }


  }


}
```

8.5 Stack

A stack is a section of memory serving for temporaty storage of data (e.g. while calculating complex expressions). Its most important task is keeping the states of significant registers during jumps to subprograms, interrupts, traps, etc. During a jump to a subprogram in this part of the memory are kept the parameter values (if any) at the time of calling a function, value of the PC register (the place reached during the execution of a program), and the frame register W14. The values of the PC and W14 registers are copied to the stack automatically, increasing the value of the W15 register by 6 (three times by 2). The compiler takes care to copy the parameters to the stack by adding a part of the code required for copying the parameters to the top-of-stack on each user call of a function.

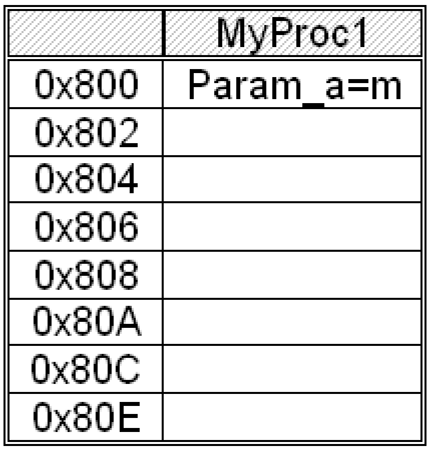
A stack is a section of memory which is usually accessed sequentially. The access is possible, of course, to any memory location, even to the locations constituting the stack, but such concept is very seldom used. An increase or a decrease of the stack, however, can be done only by a sequential access. When a datum is copied to the stack, it is pushed to the top-of-stack. Only the value from the top-of-stack can be read by the W15 register. Thus a stack is a LIFO (Last In First Out) buffer. In order to know at each moment which address is read or which address is written, one of the registers is reserved as the stack pointer register. This is the W15 register. Therefore, when a data is copied to the stack, it is written to the location pointed by the W15 register and then the W15 register is increased by 2 to point at the next free location. When a data is read from the stack, the value of the W15 register is at first decreased by 2 and then the value from the top-of-stack is read.

How does this work in practice? The following example gives a program consisting of the main program and one void type function.

Example of a call of void type function:

```
int m;




void MyProc1(int a){


int i;


  i = a+2;
}



void main(){

  TRISB = 0;

  m = 3;

  asm nop;

  MyProc1(m);

  m = 2;

}
```

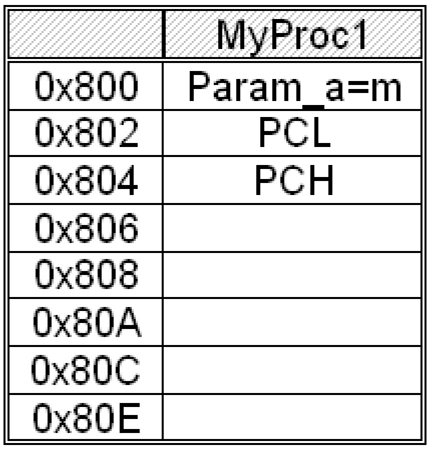
The main program begins by executing the instruction TRISB = 0;. After that, variable m is allocated the value 3. Then, the function is called. What happens at that moment? The compiler has generated a code for copying to the stack the variable m (Fig. 8-4a) and then jumps to the memory address where the subprogram is located.



	MyProc1
0x800	Param_a=m
0x802	
0x804	
0x806	
0x808	
0x80A	
0x80C	
0x80E	

Fig. 8-4a Stack before entering subprogram

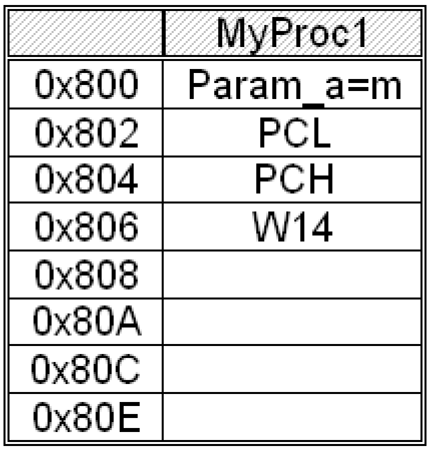
The hardware automatically copies to the stack the program register PC (Program Counter) in order to determine the location from which the execution of the program continues after the subprogram is done, Fig. 8-4b. Since the width of the PC register is 24-bit, two memory locations are required for copying the PC register. The lower 16 bits (PCL) are copied first and then the higher (PCH) 8 bits (extended to 16 bits).



	MyProc1
0x800	Param_a=m
0x802	PCL
0x804	PCH
0x806	
0x808	
0x80A	
0x80C	
0x80E	

Fig. 8-4b Stack after jump to subprogram (W15=0x806, W14=xxxx)

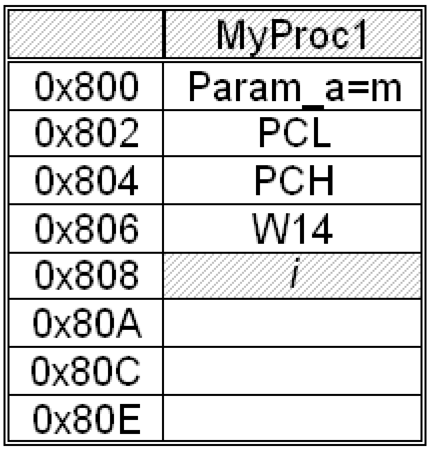
After the PC register is saved, the hardware also copies and saves the W14 register (frame register) and then writes into it the current value of the W15 register, Fig. 8-4c. This saves the information where the last location used by the main progam is. Whatever the subprogram would do with the section of the stack after this address will have no influence on the execution of the main program. Therefore, the W14 register is the boundary between the local variables of the subprogram and the parameters pushed in the stack from the main program. In addition, this allows to find the address where the parameter value valid at the time of calling the subprogram is. This is done simply by subtracting from W14 the number of locations occupied by the PC and W14 registers at the moment of jumping to the subprogram.



	MyProc1
0x800	Param_a=m
0x802	PCL
0x804	PCH
0x806	W14
0x808	
0x80A	
0x80C	
0x80E	

Fig. 8-4c Stack after entry to subprogram (W15=0x808, W14=0x808)

Upon jumping to the subprogram (void type function in this example), the compiler has the task of providing the locations required by the local variables. In this example the local variable is i of the type **int**. Fig. 8-4d shows the memory location 0x808 reserved for the local variable i. Value of the W15 register is increased to account for the reservation made for the local variables, but the value of the W14 register remains the same.



▨	MyProc1
0x800	Param_a=m
0x802	PCL
0x804	PCH
0x806	W14
0x808	▨ *i*
0x80A	
0x80C	
0x80E	

Fig. 8-4d Stack after entry to subprogram (W15=0x80A, W14=0x808)

How many locations have been occupied? The answer is three. The W14 register is 16-bit wide and occupies only one location, whereas the PC register is 24-bit wide thus it occupies two locations. The 4 bits unused for saving the PC register are used for saving the current priority level. The parameter address is claculated as W14-8. The register W14 points to the memory location next to those where the parameter, PC register, and W14 register are saved. From this value one should subtract 2 because of the W14 register, 4 because of the PC register which occupies two memory locations, and 2 because of the parameter. For this reason from the value saved in the frame register W14 one should subtract 8. The compiler has generated additional code which reserves the place at the top-of-stack occupied by the local variable i. This has not influenced the value of the register W14 but did influence the value of the register W15 because it always has to point to the top-of-stack.

After entering the function, the register W15 points at the location 0x80A, the register W14 to location 0x808, and the parameter is at W14-8=0x800. After the function is comlpeted, it is not required to check how many data has been put on the stack and how many has been taken off the stack. The register W14 points to the location next to the locations where the important registers have been saved. The value of the register W14 is written into the W15 register, previous value of the register W14 is taken off the stack and immediatley after, so is the value of the PC register. This is done automatically by the hardware. The compiler has added a section of the code to the main program which then calls the functions on the stack and reads all parameters saved in it in order to recover the value of the W15 register and return the position of the stack to the previous state.

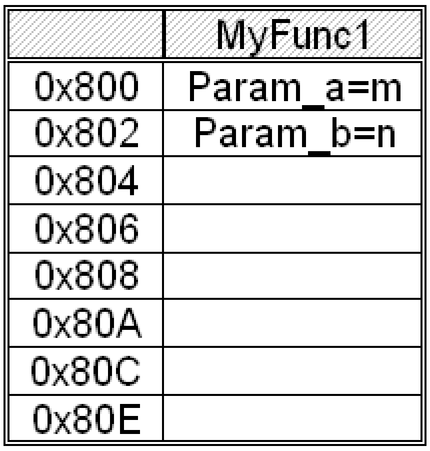
The following example gives a program consisting of one function and a program using this function. The difference between a function and a void type function lies in the result returned to the main program by the function.

Example of calling a function:

```
int m, n;




int MyFunc1(int a, int b){
```

```
int i, j;

 i = a + 2;

 j = b - 1;

 return i + j;

}



void main(){

  TRISD = 0;

  m = 3;

  n = 5;

  asm nop;

  m = MyFunc1(m, n);

  LATD = 2;

}
```

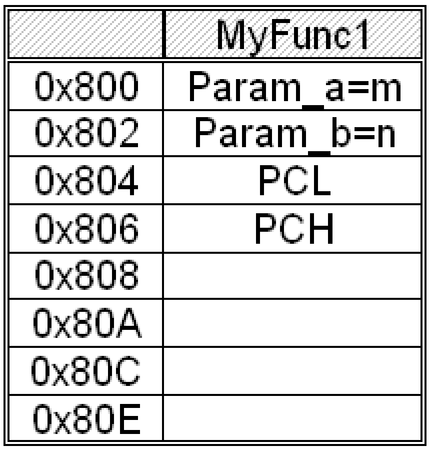
The program starts by executing the instruction **TRISD = 0**;. After that, it allocates the value 3 to the variable **m**. Then, it calls the function. What happens at this moment? The compiler has generated a code which push to the stack the variables **m** and **n**, Fig. 8-5a, and then jumps to the address in the memory corresponding to the subprogram (function).



	MyFunc1
0x800	Param_a=m
0x802	Param_b=n
0x804	
0x806	
0x808	
0x80A	
0x80C	
0x80E	

Fig. 8-5a Stack before subprogram is entered (W15=0x804, W14=xxxx)

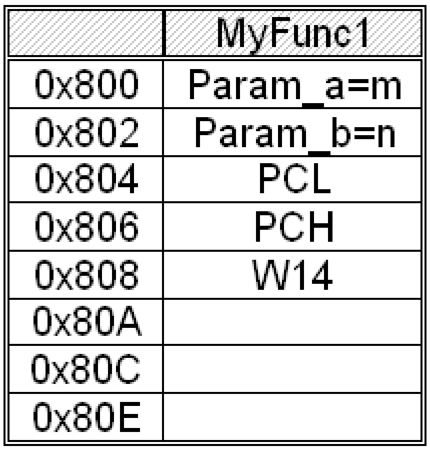
The hardware automatically saves on the stack the PC register to determine, after the subprograme is completed, the address from which the main program resumes the execution, Fig. 8-5b. The PC register is 24 bit wide, two memory locations are required in the data memory to save this register. The lower 16 bits (PCL) are copied first and then the higher (PCH) 8 bits (extended to 16 bits).



	MyFunc1
0x800	Param_a=m
0x802	Param_b=n
0x804	PCL
0x806	PCH
0x808	
0x80A	
0x80C	
0x80E	

Fig. 8-5b Stack after jump to subprogram (W15=0x808, W14=xxxx)

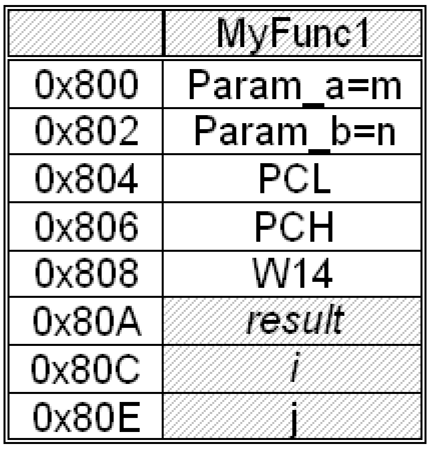
After the PC register is saved, the hardware also copies and saves the W14 register (frame register) and then writes into it the current value of the W15 register, Fig. 8-5c. This saves the information where the last location used by the main progam is. Whatever the subprogram would do with the section of the stack after this address will have no influence on the execution of the main program. Therefore, the W14 register is the boundary between the local variables of the subprogram (function in this case) and the parameters pushed in the stack from the main program. In addition, this allows to find the address where the values of the parameters a and b, valid at the time of calling the function, are. This is done simply by subtracting from W14 the number of locations occupied by the PC and W14 registers at the moment of jumping to the subprogram.



	MyFunc1
0x800	Param_a=m
0x802	Param_b=n
0x804	PCL
0x806	PCH
0x808	W14
0x80A	
0x80C	
0x80E	

Fig. 8-5c Stack after entry to subprogram (W15=0x80A, W14=0x80A)

Upon jumping to the subprogram (function in this example), the compiler has the task of providing the locations required for the result and the local variables. In this example the result is of the type int and the local variables i and j are of the type **int**. Fig. 8-5d shows the memory locations 0x80A, 0x80C, and 0x80E reserved for the rersult of the function result and the local variables i and j. Value of the W15 register is increased to account for the reservation made for the result and the local variables, but the value of the W14 register remains the same.

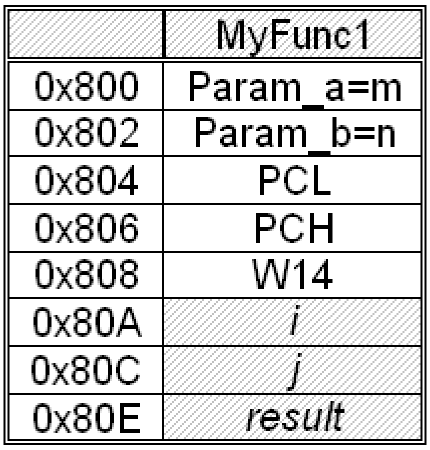


	MyFunc1
0x800	Param_a=m
0x802	Param_b=n
0x804	PCL
0x806	PCH
0x808	W14
0x80A	*result*
0x80C	*i*
0x80E	j

Fig. 8-5d Stack after entry to subprogram (W15=0x80F, W14=0x80A)

Similarly to the call of a void type function, the parameter is in the memory location W14-8, but the parameter i is in W14+2, and the result of the function (result) is written in the location W14.

What would have happened if the locations for the local variables were reserved first and then the location for the **result**? Fig. 8-5e shows the case of calling the function **MyFunc1** from the previous example if the locations for the local variables i and j were reserved first and then the location for the result.

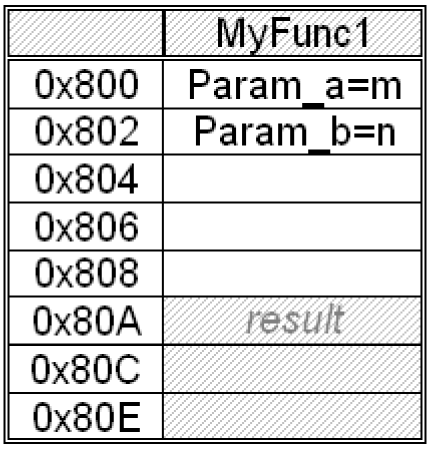


	MyFunc1
0x800	Param_a=m
0x802	Param_b=n
0x804	PCL
0x806	PCH
0x808	W14
0x80A	*i*
0x80C	*j*
0x80E	*result*

Fig. 8-5e Stack after entry to subprogram when the locations for the local variables i and j were reserved first and then the location for the result (W15=0x80C, W14=0x808)

The local variable **i** is in the memory location W14, **j** is in W14+2, and the result is in W14+4. The function could end correctly and return the result to some location. The main program could not read the result because it depends on the number of the locations reserved for the local variables. For each function the main program would have to read differently the result of the function. For this reason by reserving **first** the memory location for the result ensures that the main program knows the exact location containing the result of the function.

After completion of a subprogram (function), the content of the register W14 is written into the W15 register. This releases the memory locations of the local variables. They are of no importance for further execution of the main program. The location where the result was saved is released together wtih these locations. The process of releasing these locations does not write anything into the memory, therefore the value of the result remains the same. From the stack the previous value of the register W14 is written into the W14 register, and the values saved at the memory locations 0x806 and 0x804 are written into the PC register which points to the next instruction of the main program after the subprogram (function) was called. Only the parameters a and b remain on the stack, as shown in Fig. 8-5f. All this is done by the hardware, i.e. a microcontroller from the dsPIC30F family does that automatically. After this, the floor is taken by the compiler.



▨	MyFunc1
0x800	Param_a=m
0x802	Param_b=n
0x804	
0x806	
0x808	
0x80A	result
0x80C	▨
0x80E	▨

Fig. 8-5f Stack after return from subprogram (W15=0x804, W14=xxxx)

After return from the subprogram, on the top-of-stack are the parameters used for calling the function. Since they are no longer required, they are taken off the stack. Now the value of the stack register W15 is 0x800. The result of the function should be read. It is in the location W15+10, i.e. 0x800+0x00A=0x80A. Between the top-of-stack (pointed by the register W15) and the location where the result is saved are the memory locations where the parameters were saved (2 locations, 4 addresses) and the memory locations where the registers PC and W14 were saved (3 locations, 6 addresses). This gives the value 10 by which W15 should be increased to obtain the address of the location containing the result.

The initial value of the register W15 (pointer of the top-of-stack) is 0x800. The value of the W15 register could be changed by writing into it, but this is not recommened due to potential loss of data located on the top-of-stack.

Care should be taken that the stack is in the X space of the data memory. If the memory space is not adequately used, there is a possibility that, as the stack grows, some of the local variables are overwritten by the stack values. The probability of this type of collision increases during the exceution of DSP instructions which use the X space.

Thanks to this algorithm and the memory model, it is possible to nest the functions or call functions from the interrupt routines and traps without fear that after the return to the main program some of the registers can be overwritten and their values lost. It should be mentioned that the compiler when calling an interrupt routine, in addition to the already mentioned tasks, generates a code for saving **all** general purpose registers on the stack which ensures that while writing an interrupt routine one does not have to take into account which register was used, but the interrupt routine can be considered a program independent of the main program in so far as the general purpose registers are concerned.

The parameters and variables which take more than one memory location, e.g. 32-bit integers, are kept in the memory in the following way. The lower 16 bits are saved first (in the lower address) and then the higher 16 bit are saved (in the higher address).

At the end of this section a description is given of the registers used to control modulo and bit-reversed addressing.

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
MODCON	0X0046	XMODEN	YMODEN	-	-	BWM<3:0>				YWM<3:0>				XWM<3:0>				0x0000

Table 8-3 MODCON register

```
XMODEN - X space modulo addressing enable bit
YMODEN - Y space modulo addressing enable bit
BWM<3:0> - Register select for bit-reversed addressing bits
YWM<3:0> - Y space register select for modulo addressing bits
XWM<3:0> - X space register select for modulo addressing bits
```

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
XMODSRT	0X0048	XS<15:1>															0	0x0000

Table 8-4 XMODSRT register

```
XS<15:1> - X space modulo addressing start address bits
```

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
XMODEND	0X0050	XE<15:1>															1	0x0001

Table 8-5 XMODEND register

```
XE<15:1> - X space modulo addressing end address bits
```

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
YMODSRT	0X0052	YS<15:1>															0	0x0000

Table 8-6 YMODSRT register

```
YS<15:1> - Y space modulo addressing start address bits
```

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
YMODEND	0X0054	YE<15:1>															1	0x0001

Table 8-7 YMODEND register

```
YE<15:1> - Y space modulo addressing end address bits
```

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
XBREV	0X0056	BREN	XB<14:1>															0x0000

Table 8-8 XBREV register

```
BREN -  Bit-reversed addressing enable bit
XB<14:1> - Bit-reversed modifier bits
```

8.6 Addressing modes

Usually the programs for the dsPIC devices are written in some of the higher programming languages. However, sometimes it is required that a section of the code is written in the assembler. This need is particularly emphasized for the processes involving complex processing. The optimization is possible in some of the higher programming languages, but it is most efficient when using the assembler, assuming that the architecture of the dsPIC30F devices is known. In these cases the core of the algorithm (most demanding part of the program) is programmed using the assembler. The basic problem when using the assembler instruction set is data memory access, which could be done in several ways. Each of these has its merits and defficiencies. It is thus very important to know the addressing modes and use them correctly while writing a program.

Each assembler instruction can be divided into two parts. The first part of an instruction is the operation which is carried out (like **mov**, **add**, etc.) and the second is the operand(s). The operand is a value undergoing the operation. E.g. the instruction **DEC W0, W1** carries out the operation of decrementing, defined by DEC, and the operands are the values in the registers **W0** and **W1**.

For the family of dsPIC30F devices there are four addressing modes:

- direct memory addressing,
- direct register addressing,
- indirect register addressing, and immediate adrressing.

8.6.1 Direct memory addressing

Direct memory addressing is the mode where the operand is at the memory location specified by the instruction. This means that the operation is accompanied by the address in the memory where the value undergoing the operation is located. An example of this mode can be when the task is to take the value located in an addess in the memory and transfer it to a general purpose register (W0...W15).

Example:

```
MOV 0x0900, W0.
```

This defines the operation of moving MOV (MOVe) while the operands are the value at the adrress 0x0900 in the data memory and the value kept in the register W0. The contents of the data memory at location 0x0900 and the register W0 before and after this instruction are shown in Fig. 8-5.

before

0x900	0x0123
W0	0x000

after

0x900	0x0123
W0	0x0123

Fig. 8-5 Contents of the data memory at location 0x0900 and the register W0 before and after the instruction

When using direct memory addressing, there are restrictions depending on the type of the operation used.

The majority of operations can use direct memory addressing only for the lowest 8KB of the memory, while some (like MOV) can use all 64KB. The operation MOV therefore can access any memory location and read the value from it or write in a new value.

The second restriction is that during execution of one instruction only one memory location is accessible (the exception are DSP operations to be discussed later). In an istruction having several operands only one can be addressed by direct memory addressing.

Direct memory addressing supports 8-bit and 16-bit access to the memory. From the memory can be read or in the memory can be written in an 8-bit (byte) or a 16-bit (basic size for the dsPIC30F family) value.

Direct memory addressing is most often used when a value shoud be read from the memory or a result saved in it.

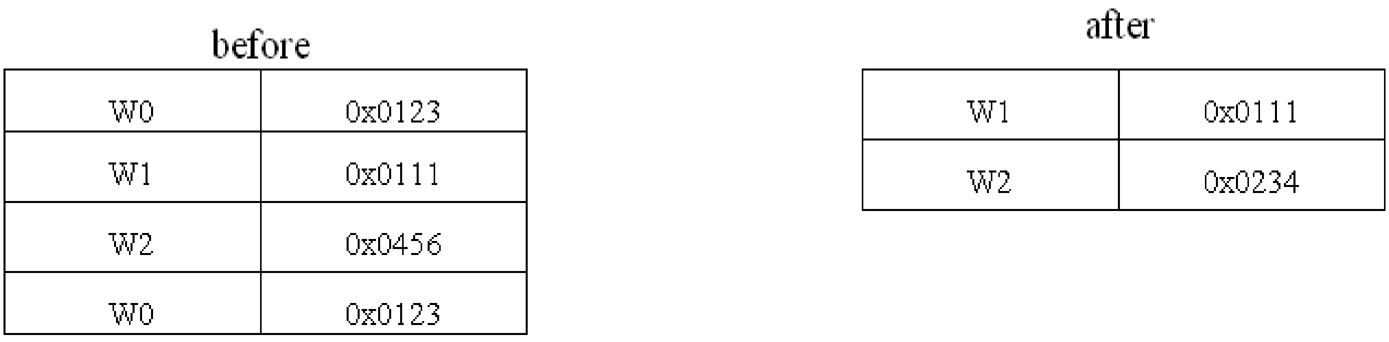
8.6.2 Direct register addressing

Direct register addresing is the addressing mode where the operand is in the register cited in the instruction. Any general purpose register (W0...W15) can be used.

Example:

```
ADD W0, W1, W2
```

The consequence of this instruction will be taking the values from the registers W0 and W1, adding them, and saving the result in the register W2. Here there are three operands: the values of W0, W1, and W2. The third operand will be overwritten by the result of adding of the first two. The values of the registers W0, W1, and W2 before and after the instruction are shown in Fig.8-6.



before

W0	0x0123
W1	0x0111
W2	0x0456
W0	0x0123

after

W1	0x0111
W2	0x0234

Fig. 8-6 Contents of the registers W0, W1, and W2 before and after the instruction

The advantage of direct register addressing over direct memory addressing is that there are no restrictions in its use. All operations support direct register addressing. For this reason direct register addressing is most often used; it is also very suitable for looping (**DOO** and **LOOP**).

8.6.3 Indirect register addressing

Indirect register addressing means that the operand is in the memory location whose address is written in one of the general purpose registers (W0...W15). The value in the register in this case is the pointer to the memory location where the operand is saved. Indirect register addressing is very useful because it allows the value in the register to be changed before or after the operation is carried out (within the same instruction). This allows that the data, saved sequentially in the memory (one after the other), are processed very efficiently.

Example:

```
MOV [W1], W3
```

The consequence of this instruction is that the value in the memory pointed by the register W1 will be written in the register W2. The register values and memory locations before and after the instruction are shown in Fig. 8-7.

before

W1	0x0900
W2	0x0543
0x0900	0x1673

after

W0	0x0900
W1	0x1673
W2	0x1673

Fig. 8-7 Values of the registers and memory locations before and after the instruction

The flow of operations is the following:

1. The value in the register W1 is read.
2. The value in the memory location pointed by the register W1 is read,
3. The value read from the memory is written in the register W2.

As already mentioned, indirect register addressing allows processing several data, sequentially saved in the memory. This is accomplished by changing automatically the value of the register used for indirect register addressing.

There are four methods of changing the register value in the process of indirect register addressing:

- pre-increment [++W1],
- pre-decrement [--W1],
- post-increment [W1++],
- post-decrement [W1--].

In the pre-increment and pre-decement addressing modes the value of the register is changed (increased or decreased) first and then the operand address is read. In the post-increment and post-decrement addressing modes the value of the register is read first and then changed (increased or decreased). In this way the register values are increased or decreased in two steps. The reason that there are two steps is because 16-bit words are in question. Of course, it is possible to read only one byte, but it should be specified that an 8-bit operation is being executed. Otherwise, 16-bit operation is understood.

Example:

```
MOV $0900, W1

MOV #0, W2

REPEAT #5

MOV W2, [W1++]
```

This example writes zeros to six sequential locations in the memory. The instruction REPEAT has the consequence that the subsequent operation is executed the specified number of times plus one, i.e. MOV W2, [W1++] will be executed six times.

Attention!!!
All operations are 16-bit, unless specified otherwise. This means that one memory location contains two bytes. Even if the operation was 8-bit, the value of the register would be incremented by two, not by one.

The values of all relevant registers and memory locations before the execution of the above program, before the loop (after the first two instructions) and after the execution of the loop are shown in Fig. 8-8.

At Start	Before the loop	After the loop
W1 0xFFFF	W1 0x0900	W1 0x0900
W2 0xFFFF	W2 0x0000	W2 0x0000
0x0900 0xFFFF	0x0900 0xFFFF	0x0900 0x0000
0x0902 0xFFFF	0x0902 0xFFFF	0x0902 0x0000
0x0904 0xFFFF	0x0904 0xFFFF	0x0904 0x0000
0x0906 0xFFFF	0x0906 0xFFFF	0x0906 0x0000
0x0908 0xFFFF	0x0908 0xFFFF	0x0908 0x0000
0x090A 0xFFFF	0x090A 0xFFFF	0x090A 0x0000

Fig. 8-8 The values of all relevant registers and memory locations before, during and after the execution of the program

There is another mode of indirect register addressing. This is the shift register mode. It is very useful for accessing members of an array.

Example:

MOV [W1+W2], W3

The address of the first operand is calculated by adding the values in the registers W1 and W2. The obtained value is the address in the memory where the operand is saved (the value which should be written into the register W3). The location with this address is read and the value written into the register W3. All this is performed in one instruction.

Of all described modes of indirect register addressing (without register modification, with register modification, and shift register), the majority of instructions supports only indirect rgeister addressing without modification. The instruction MOV supports all the modes. DSP instruction set supports only post-increment and post-decrement mode, but the value by which a register is modified can be selected. The increment/decrement value can, in this case, be ±2, ±4, and ±6. For more details concerning DSP instructions, see **Chapter 11**.

8.6.4 Literal adrressing

Literal addressing is the addressing mode where the operand is located immediately after the operation. It is not required to read any register or memory location. The operand is carried together with the operation code as a constant to be used during the execution of the instruction.

The size of the constant depends on the operation to be executed. The constant can be signed or unsigned and can be saved with a different number of bits. It is customary to specify the limitations in one of the following ways:

- #lit4, which specifies a 4-bit unsigned constant. This means that the range of values of the constant is 0...15. The last number denotes the number of bits and Lit denotes that the constant has no sign.
- #bit4, which specifies a 4-bit unsigned constant. The difference between #lit4 and #bit4 is that #bit4 denotes bit position within a word. It is used in the instructuions setting certain bit to logic zero or logic one (BCLR, BSET, BTG,...).
- #Slit4, which specifies a signed 4-bit constant. The range of #Slit4 is from –8 to +7.

Table 8-9 gives a list of all possible formats of the constant for literal addressing, together with the instructions where the constant is used. These are the assembler instructions which are very seldom used if the programming is performed in a higher level language, but it is of considerable importance to know the limitations in order to use correctly certain instructions of the higher level languages. E.g. shift operand (instructions **ASR, LSR, SL**) can be done within the range 0-16, which is logical since the 16-bit data are involved.

Operand	**Instruction where it is used**	**Range**
#bit4	BCLR, BSET, BTG, BTSC, BTSS, BTST, BTST.C, BTST.Z, BTSTS, BTSTSS.C, BTSTS.Z	0 ... 15
#lit1	PWRSAV	0 ... 1
#lit4	ASR, LSR, SL	0 ... 15
#lit5	ADD, ADDC, AND, CP, CPB, IOR, MUL.SU, MUL.UU, SUB, SUBB, SUBBR, SUBR, XOR	0 ... 31
#lit8	MOV.B	0 ... 255
#lit10	ADD, ADDC, AND, CP, CPB, IOR, RETLW, SUB, SUBB, XOR	0 ... 1023
#lit14	DISI, DO, LNK, REPEAT	0 ... 16383
#lit16	MOV	0 ... 65535
#Slit4	ADD, LAC, SAC, SAC.R	-8 ... +7
#Slit6	SFTAC	-32768 .. +32767

#Slit10	MOV	-512 ... +512

Table 8-9 Immediate addressing operands

Example:

```
ADD W1, #4, W2
```

In the example the value of the register W1 is added 4 and the result is written into W2. For the second operand the literal addressing is used. From the table it can be seen that with the instruction ADD one can use constants within the range 0...31.

NOTE: Individual instructions can use different ranges. E.g. the instruction ADD has three forms for literal addressing. This should be taken care of only while writing a part of the program using the assembler. When using higher programing languages (PASCAL, C, BASIC), the compiler takes care of the form that should be alocated to a given instruction.

Chapter9: SPI Module

Introduction

The Serial Peripheral Interface (SPI) module is a synchronous serial interface useful for communicating with other peripheral or microcontoller devices. The examples of the peripheral devices are: serial EEPROMs, shift registers, display drivers, serial A/D converters, etc. The SPI module is compatible with Motorola’s SPI and SIOP interfaces.

Depending on the variant, the dsPIC30F family offers one or two SPI modules on a single device. E.g. dsPIC30F3014 has one SPI interface module, whereas dsPIC30F6014A has two.

A standard SPI serial port consists of the following special function registers (SPR):

- SPIxBUF – SFR used to buffer data to be tramsittesd and data that have been received. It consists of two memory locations SPIxTXB (data trasmit) and SPIxRXB (data receive).
- SPIxCON – a control register that configures the module for various modes of operation.
- SPIxSTAT – a status register that indicates various status conditions.

In addition, there is a 16-bit register, SPIxSR, that is not memory mapped. It is used for shifting in and out of the SPI port.

The memory mapped special function register SPIxBUF, the SPI data receive/transmit register, actually consists of two separate registers – SPIxTXB and SPIxRXB. If a user writes data to the SPIxBUF address, internally the data are written to the SPIxTXB (transmit buffer) register. Similarly, when the user reads the received data from the SPIxBUF, internally the data are read from the SPIxRXB (receive buffer) register. This double buffering of transmit and receive operations allows continuous data transfers in the background.

Attention!!!
The user can not write to the SPIxTXB register or read from the SPIxRXB register directly. All reads and writes are performed on the SPIxBUF register.

Fig. 9-1 shows functional block diagram of the SPI module. In addition to the above registers, the SPI module serial interface consists of the following four pins:

- SDx – serial data input,
- SDOx – serial data output,
- SCKx – shift clock input or output,
- SSx – active low slave select or frame synchronization I/O pulse.

NOTE: The SPI module can be configured to operate using 3 or 4 pins. In the 3-pin mode, the SSx pin is not used.

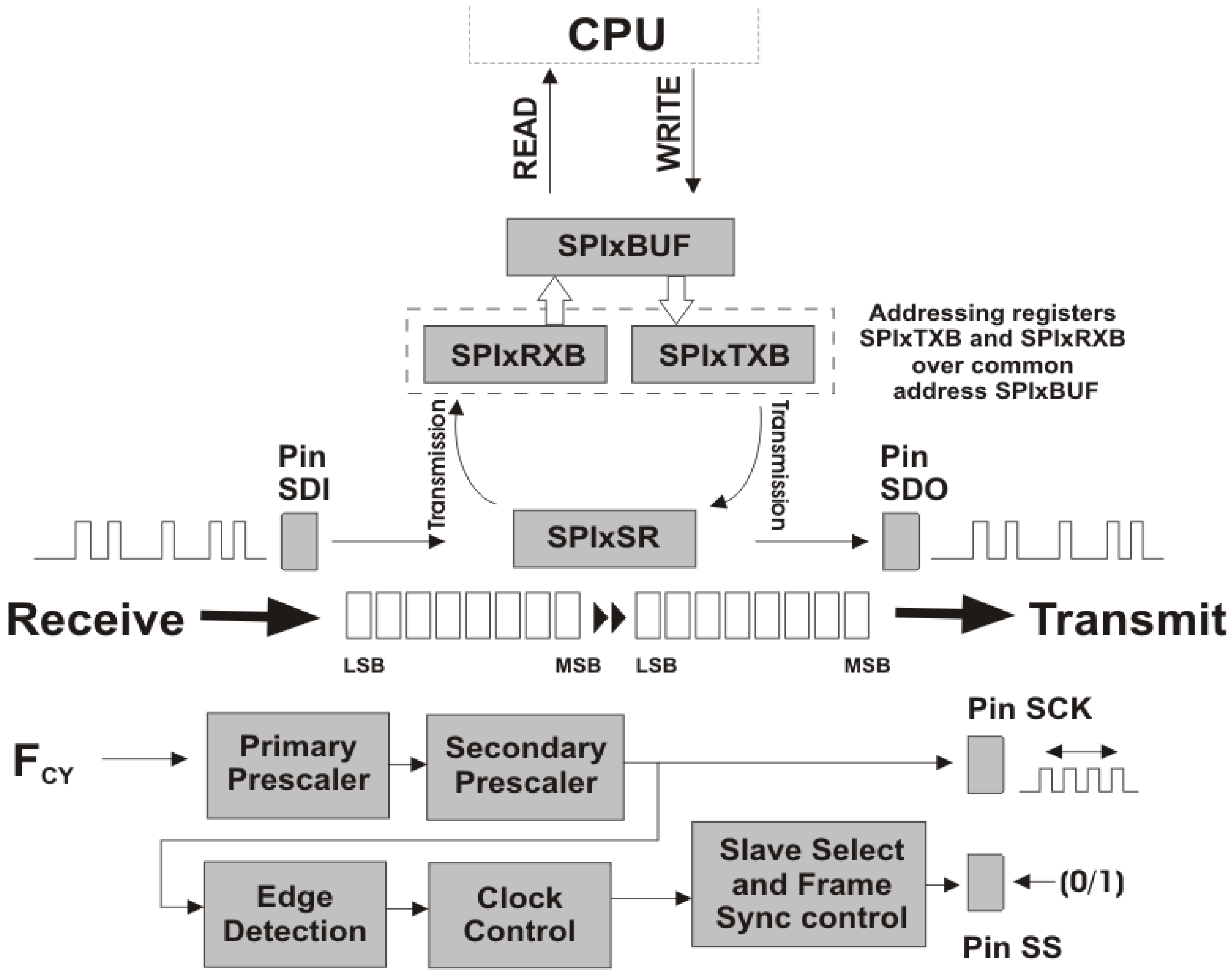


Fig. 9-1 Functional block diagram of SPI module

The SPI module has the following flexible operating modes:

- 8-bit and 16-bit data transmission/reception,
- Master and slave modes,
- Framed SPI modes.

9.1 8-bit and 16-bit data transfer

A control bit MODE16 (SPIxCON<10>) allows the module to communicate in either 8-bit or 16-bit modes. The functionality will be the same for each mode except the number of bits that are received and transmitted. The following should be noted in this context:

- The module is reset when the value of the MODE16 control bit is changed. Consequently, the bit should not be changed during normal operation.
- Data are transmitted out of bit 7 of the shift register SPIxSR for 8-bit operation and out of bit 15 for 16-bit operation. In both modes data are shifted into bit ‘0’ of the SPIxSR.
- 8 clock pulses at the SCKx pin are required to shift in/out data in 8 bit mode, while 16 clock pulses are required in the 16-bit mode.

9.2 Master and slave modes

In a multi-processor operation when the SPI interface is used, the microcontrollers operate in the master and slave modes. In master mode the microcontroller is in full control of the communication since it initiates and ends the communication session and generates the SPI clock signal. In slave mode the microcontroller listens when the master initiates and ends the communication session and uses the SPI clock signal generated by the master. SPI master/slave connection is shown in Fig. 9-2. The figure shows the 4-pin SPI interface even though this connection can also operate with the 3-pin SPI interface.

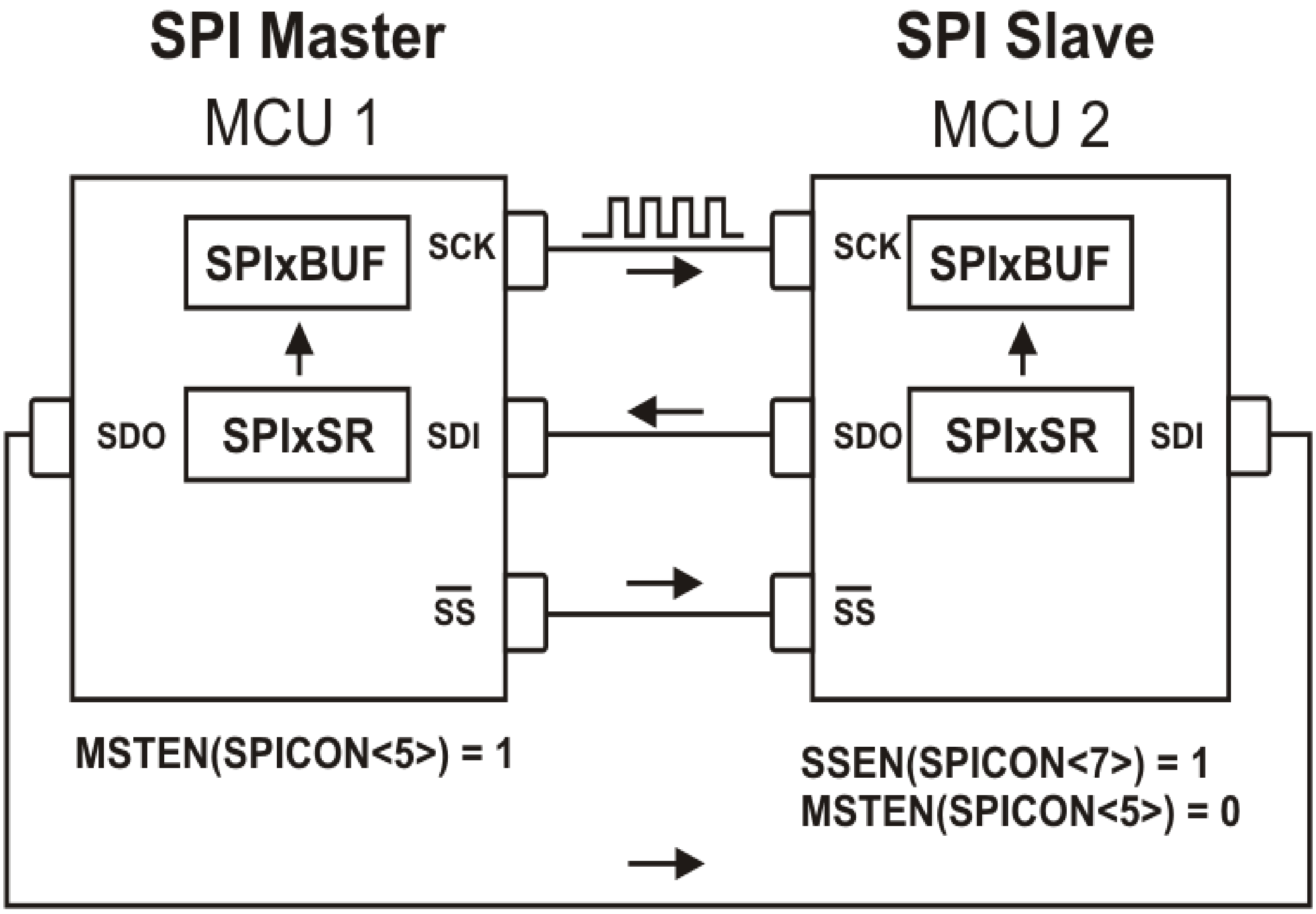


Fig. 9-2 Master/slave connection using SPI interface

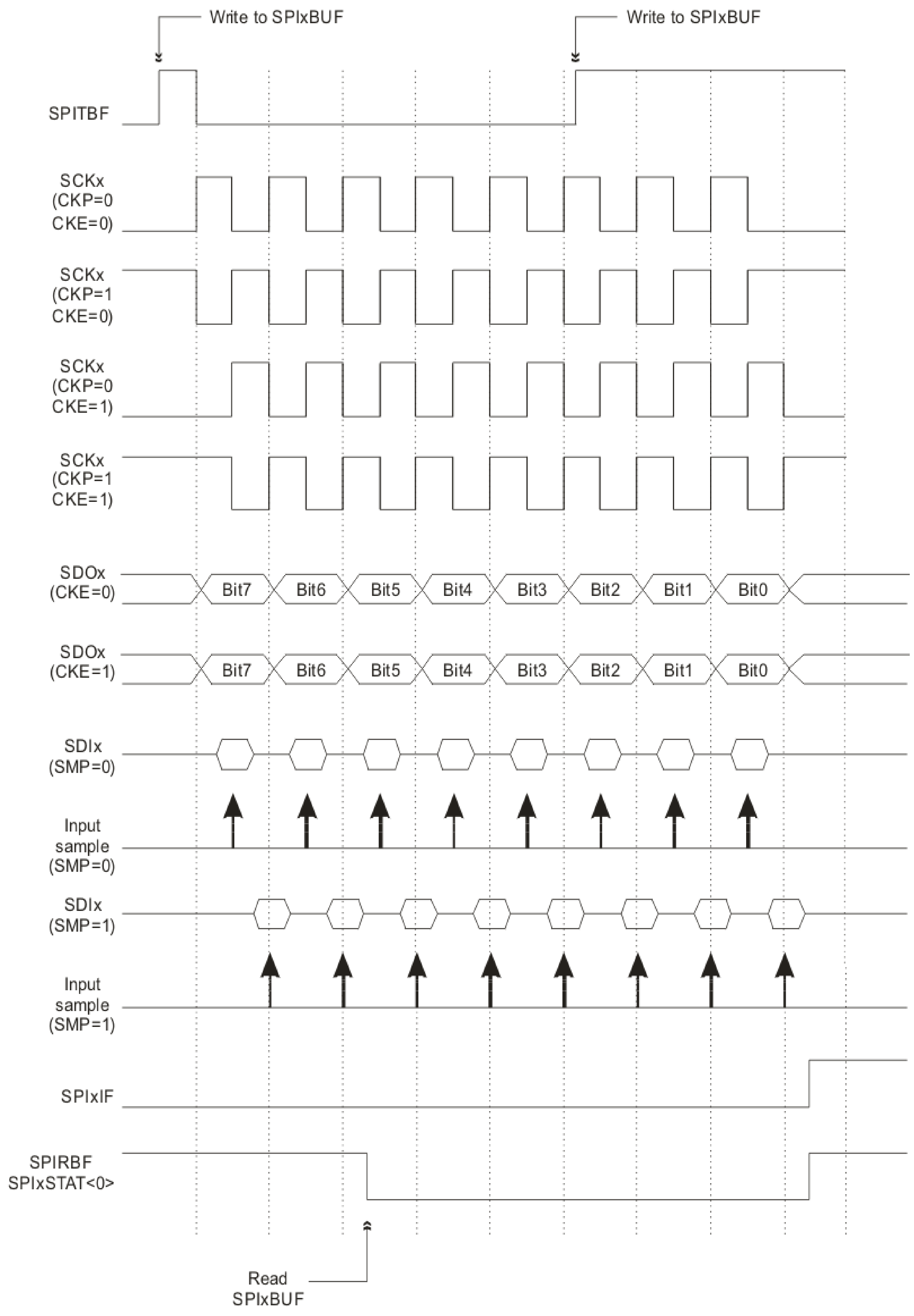
NOTE: Using the SSx pin in slave mode of operation is optional.

9.2.1 Master Mode

The following steps should be taken to set up the SPI module for the master mode of operation:

1. If using interrupts:
    1. Clear the SPIxIF bit in the respective IFSn register,
    2. Set the SPIxIE bit in the respective IECn register,
    3. Write the SPIxIP bits in the respective IPCn register.
2. Write the desired settings to the SPxCON register with MSTEN (SPIxCON<5>) =1.
3. Clear the SPIROV (SPIxSTAT<6>) status bit.
4. Enable SPI operation by setting the SPIEN (SPIxSTAT<15>) control bit.
5. Write the data to be transmitted to the SPIxBUF register. Transmission will start as soon as data are written to the SPIxBUF register.

In master mode, the system clock is prescaled and then used as the serial clock. The prescaling is based on the settings in the PPRE<1:0> (SPIxCON<1:0>) and SPRE<1:0> (SPIxCON<4:2>) control bits. The serial clock generated in the master device is via the SCKx pin sent to slave devices. Clock pulses are only generated when there are data to be transmitted/received. The CKP (SPIxCON<6>) and CKE (SPIxCON<8>) control bits determine on which edge of the clock data transmission occurs.



Write to SPIxBUF
Write to SPIxBUF
SPITBF
SCKx (CKP=0 CKE=0)
SCKx (CKP=1 CKE=0)
SCKx (CKP=0 CKE=1)
SCKx (CKP=1 CKE=1)
SDOx (CKE=0)
Bit7 Bit6 Bit5 Bit4 Bit3 Bit2 Bit1 Bit0
SDOx (CKE=1)
Bit7 Bit6 Bit5 Bit4 Bit3 Bit2 Bit1 Bit0
SDIx (SMP=0)
Input sample (SMP=0)
SDIx (SMP=1)
Input sample (SMP=1)
SPIxIF
SPIRBF SPIxSTAT<0>
Read SPIxBUF

Fig. 9-3 SPI module in master mode of operation

With the help of Fig. 9-3, the following description of the SPI module operation in master mode can be given.

1. Once the module is set up for master mode of operation and enabled, data to be transmitted are written to the SPIxBUF register, i.e. to the SPIxTXB register, the sequence of transmitting via the SPI transmitter starts. The presence of new data in the SPIxBUF register, i.e. in the SPIxTXB buffer, is denoted by setting the SPITBF status bit (SPIxSTAT<1>).
2. The contents of the SPIxTXB register are moved to the shift register, SPIxSR, and the SPITBF bit is cleared by the module.
3. A series of 8/16 clock pulses shifts out 8/16 bits of transmit data from the SPIxSR to the SDOx pin and simultaneously shifts in the data at the SDIx input pin into the SPIxSR register of the slave device.
4. When the transfer is complete, the following events will occur:
    1. The interrupt flag bit, SPIxIF, is set. SPI interrupts can be enabled by setting the interrupt enable bit SPIxIE. The SPIxIE flag is not cleared automatcally by the hardware.
    2. Also, when the ongoing transmit and receive operation is completed, the contents of the SPIxSR are moved to the SPIxRXB register.
    3. The SPIRBF (SPIxSTST<0>) is set by the module, indicating that the receive buffer is full. Once the SPIxBUF register is read by the used code, the hardware clears the SPIRBF bit.
5. If the SPIRBF bit is set (receive buffer is full) when the SPI module needs to transfer data from SPIxSR to SPIxRXB, the module will set the SPIROV (SPIxSTST<6>) status bit, indicating an overflow condition.
6. Data to be transmitted can be written to SPIxBUF by the user software at any time as long as the SPITBF (SPIxSTST<1>) status bit is clear. The write can occur while SPIxSR is shifting out the previously written data, allowing continuous transmission.

9.2.2 Slave mode

The following steps should be taken to set up the SPI module for the slave mode of operation:

1. Clear the SPIxBUF register.
2. If using interrupts:
    1. Clear the SPIxIF bit in the respective IFSn register,
    2. Set the SPIxIE bit in the respective IECn register,
    3. Write the SPIxIP bits (priority level) in the respective IPCn register.
3. Write the desired settings to the SPIxCON with MESTEN (SPIxCON<5>)=0.
4. Clear the SMP (SPIxCON<9>) control bit. This specifies that input sampling is performed in the middle of the SPI clock.
5. If the CKE (SPIxCON<8>) bit is set, then the SSEN (SPIxCON<7>) control bit must be set, thus enabling 4-pin serial interface.
6. Clear the SPIROV (SPIxSTAT<6>) bit and, finally, enable SPI operation by setting the SPIEN (SPIxSTAT<15>) control bit.

In slave mode, data are transmitted and received as the external clock pulses appear on the SCKx pin. The CKP (SPIxCON<6>) and CKE (SPIxCON<8>) control bits determine on which edge of the clock data transmission occurs. Both data to be transmitted and data that are received are respectively written into or read from the SPIxBUF register.

A few additional features provided in the slave mode are:

- **Slave select synchronization –** the SSx pin allows a synchronous slave mode. If the SSEN (SPIxCON<7>) control bit is set, transmission and reception is enabled in slave mode only if the SSx pin is driven to a low state (pin SSx is active if at the low logic level). If the SSEN bit is set and the SSx pin is driven high, the SDOx pin is no longer driven and the current transmission stops even if the module is in the middle of a transmission. An aborted transmission will be retried the next time the SSx pin is driven low. If the SSEN (SPIxCON<7>) control bit is cleared, the SSx pin does not affect the SPI interface (three-pin SPI interface).
- **SPITBF status flag operation –** the function of the SPITBF (SPIxSTAT<1>) bit is different in the slave mode od operation. The following describes the function of the SPITBF for various settings of the slave mode of operation:
    1. If SSEN (SPIxCON<7>) is cleared (3-pin SPI interface), the SPITBF is set when the SPIxBUF (i.e. SPIxTXB) is loaded by the user code. It is cleared when the module transfers SPIxTXB to the shift register SPIxSR. This is similar to the SPITBF bit function in master mode.
    2. If SSEN (SPIxCON<7>) is set (4-pin SPI interface), the SPITBF is set when the SPIxBUF (i.e. SPIxTXB) is loaded by the user code. It is cleared only when the module completes data transmission. A transmission will be aborted when the SSx pin goes high and may be retried at a later time when the pin goes low. Each data word is held in SPIxTXB until all bits are transmitted to the receiver.

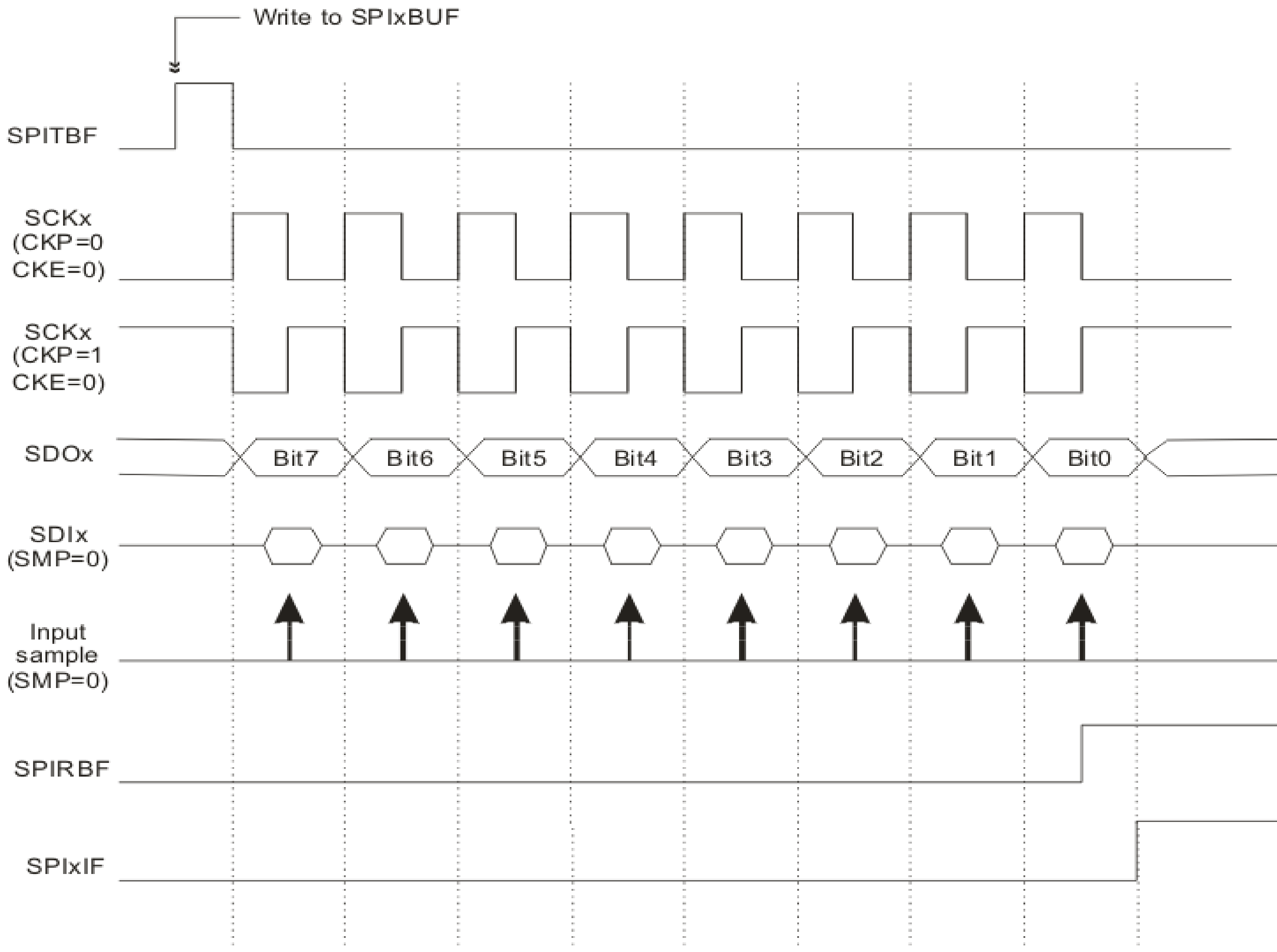


Fig. 9-4 SPI slave mode: 3-pin SPI interface

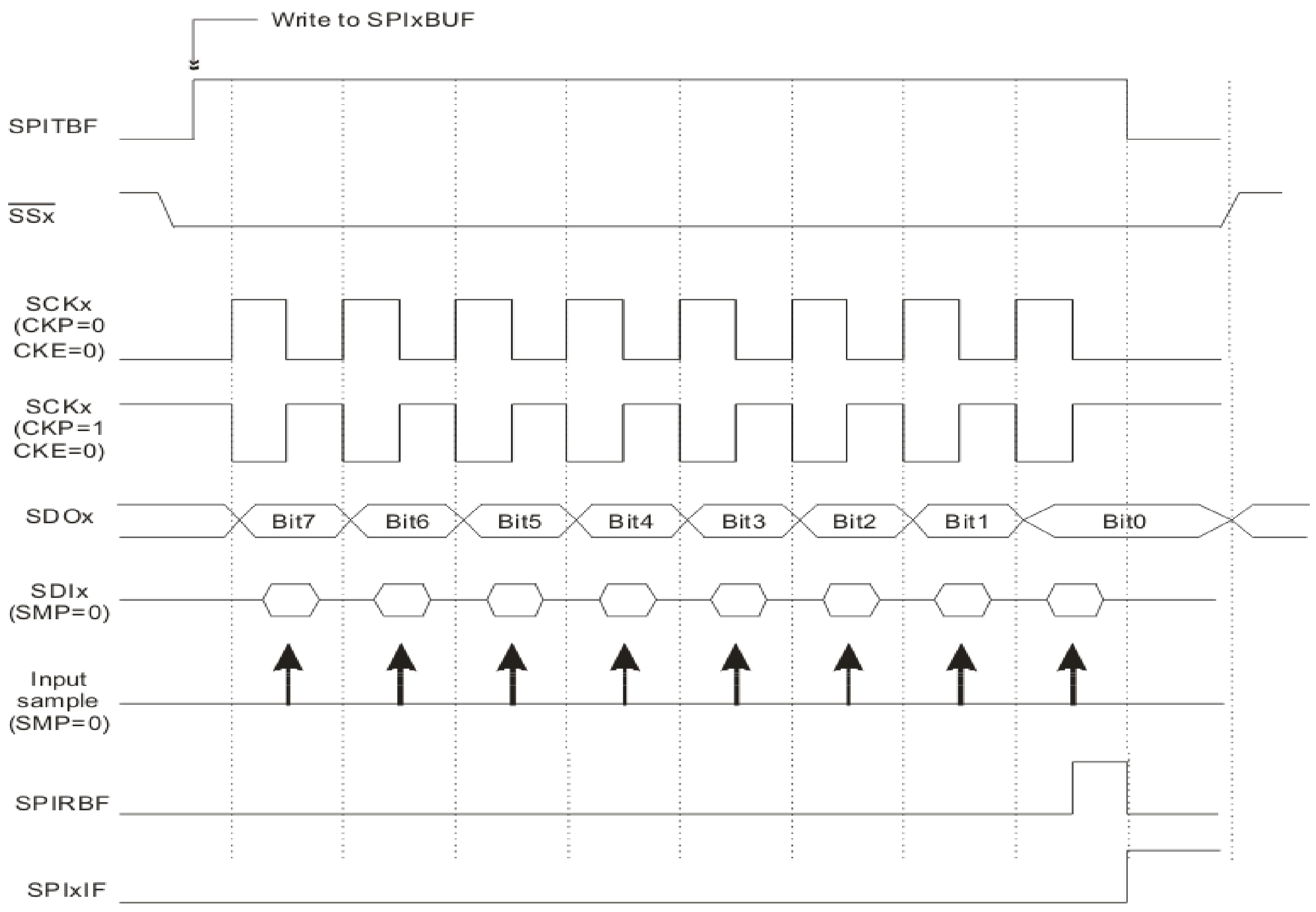


Fig. 9-5 SPI slave mode: 4-pin SPI interface

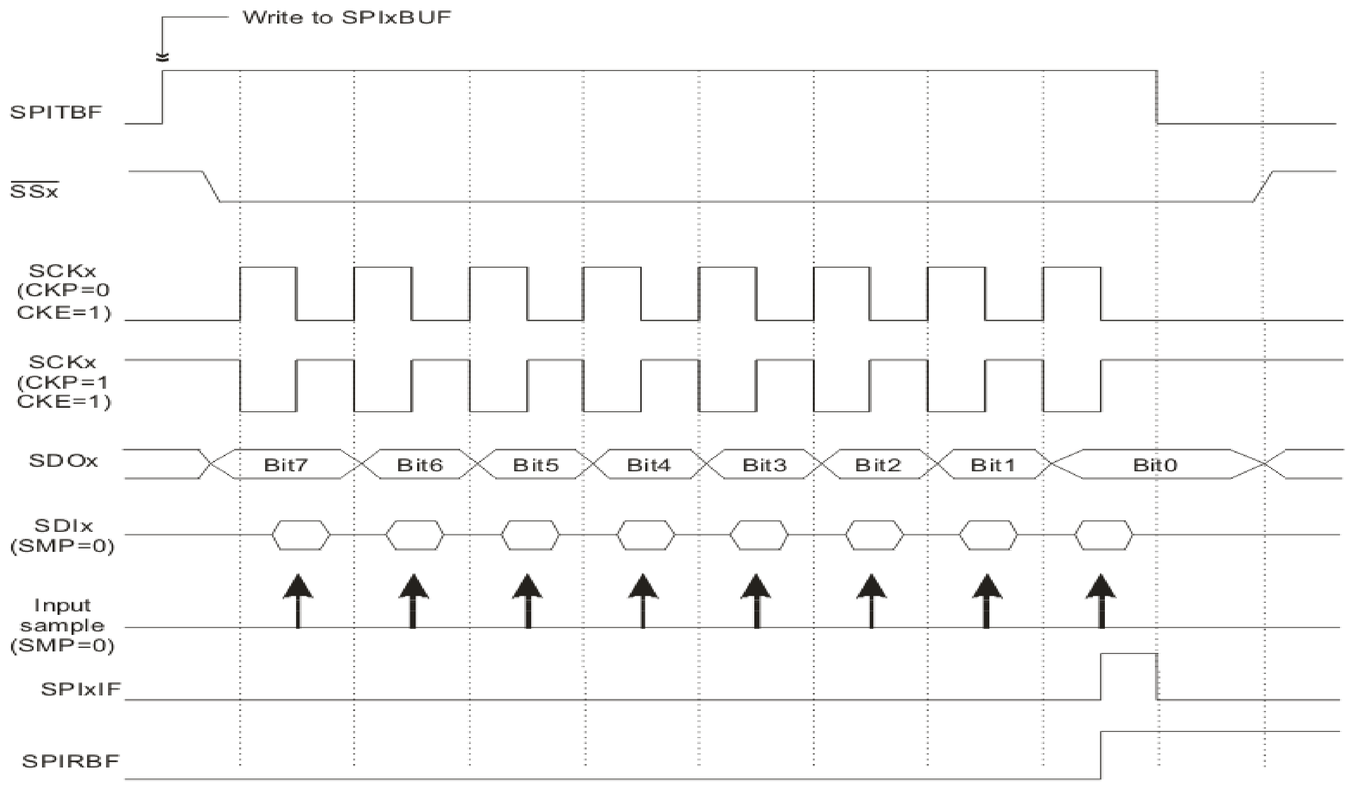


Fig. 9-6 CKP and CKE bit functionality

The operation for 8-bit mode is shown. The 16-bit mode is similar.

When a new data word has been shifted into SPIxSR and the previous contents of SPIxRXB have not been read by the user software, the SPIROV bit (SPIxSTAT<6>) will be set (overflow condition). The module will not transfer the received data from SPIxSR to SPIxRXB. Further data reception is disabled until the SPIROV bit is cleared. The SPIROV bit is not cleared automatically by the module and **must be cleared by the user software**.

Setting the control bit DISSDO (SPIxCON<11>) disables transmission at the SDOx pin. This allows the SPIx module to be configured for a receive only mode of operation. If the DISSDO bit is set, the SDOx pin will be controlled by the respective port function (input or output).

9.3 Framed SPI modes

The module supports a very basic framed SPI protocol while operating in either master or slave modes. The following features are provided in the SPI module to support framed SPI modes:

1. The control bit FRMEN (SPIxCON<14>) enables framed SPI modes and causes the SSx pin to be used as a frame synchronization pulse input or output pin,
2. The control bit SPIFSD (SPIxCON<13>) determines whether the SSx pin is an input or an output, i.e. whether the module receives or generates the frame synchronization pulse.
3. The frame synchronization pulse is an active high pulse for a single SPI clock cycle.

The following two framed SPI modes are supported by the SPI module:

1. Frame master mode: the SPI module generates the frame synchronization pulse and provides this pulse to other devices at the SSx pin.
2. Frame slave mode: the SPI module uses a frame synchronization pulse received at the SSx pin.

The framed SPI modes are supported in conjunction with the master and slave modes. The following four framed SPI configurations are available to the user: SPI master mode and frame master mode, SPI master mode and frame slave mode, SPI slave mode and frame master mode, and SPI slave mode and frame slave mode. These four modes determine whether or not the SPIx module generates the serial clock and the frame synchronization pulse. Fig. 9-7 shows block diagram of the master and slave connection in master and frame slave mode.

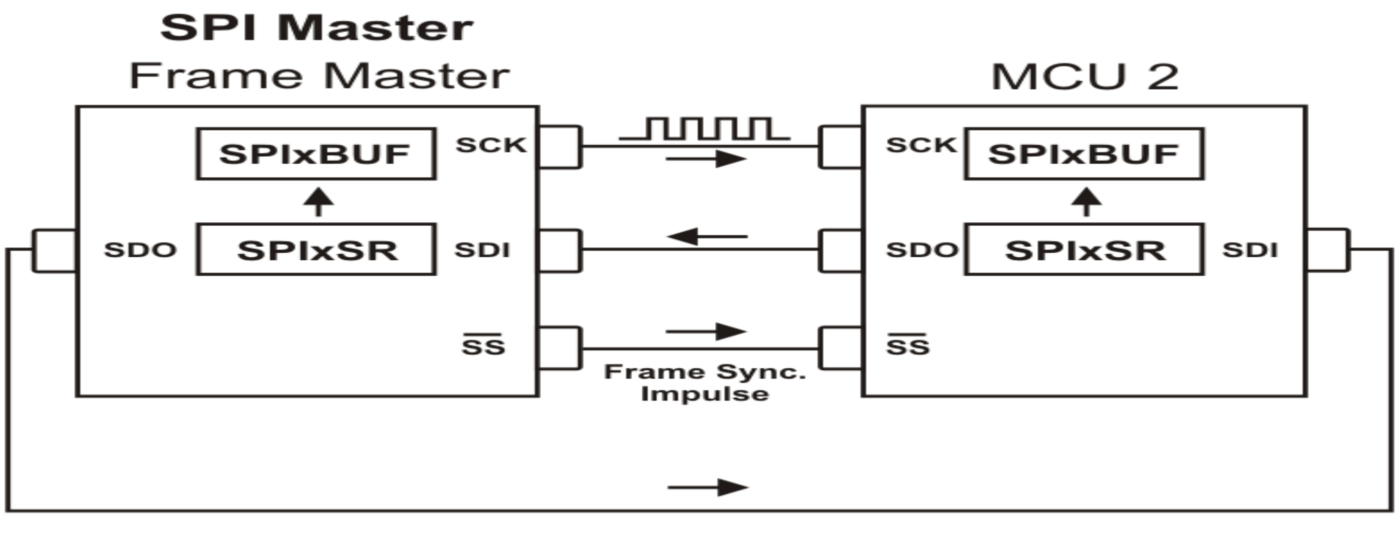


Fig. 9-7 Master and slave connection in master and frame slave mode

The SPI clock at the SCKx pin is controlled by the FRMEN (SPIxCON<14>) and MSTEN (SPIxCON<5>) control bits. When FRMEN (SPIxCON<14>)=1 and MSTEN (SPIxCON<5>)=1, the SCKx pin becomes an output and the SPI clock at SCKx becomes a free running clock, i.e. it will exists irrespective of whether the module is transmitting data or waiting. This mode is intended for the master devices. When FRMEN (SPIxCON<14>)=1 and MSTEN (SPIxCON<5>)=0, the SCKx pin becomes an input pin. This mode is intended for the slave devices.

The polarity of the clock pulse is selected by the CKP (SPIxCON<6>) control bit. The CKE (SPIxCON<8>) control bit is not used for the framed SPI modes and should be cleared by the user software. When CKP (SPIxCON<6>)=0, the frame synchronization pulse output and the SDOx output change on the rising edge of the clock pulses at the SCKx pin. Input data are sampled at the SDOx input pin on the falling edge of the SPI clock pulses at the SCKx pin.

When the control bit CKP (SPIxCON<6>)=1 the frame synchronization pulse output and the SDOx data output change on the falling edge of the clock pulses at the SCKx pin. Input data are sampled at the SDIx input pin on the rising edge of the SPI clock at the SCKx pin.

When the SPIFSD (SPIxCON<13>) control bit is cleared, the SPIx module is in the frame master mode of operation. In this mode the frame synchronization pulse is initiated by the module when the user software writes the transmit data to SPIxBUF location, i.e. writing the SPIxTXB register with transmit data. At the end of the frame synchronization pulse, the SPIxTXB is transferred to the SPIxSR and data transmission/reception begins.

When the SPIFSD (SPIxCON<13>) control bit is set, the module is in frame slave mode. The frame synchrinization pulse is generated by an external source. When the module samples the frame synchronization pulse, it will transfer the contents of the SPIxTXB register to the SPIxSR register and data transmission/reception begins. The user must make sure that the correct data are loaded into the SPIxBUF for transmission before the frame synchronization pulse is received.

Attention!!!
Receiving a frame synchronization pulse will start a transmission, regardless of whether data were written to SPIxBUF. If no write was performed, the old contents of SPIxBUF will be transmitted.

9.3.1 SPI module in master mode and frame master mode

This framed SPI mode is enabled by setting the MSTEN (SPIxCON<5>) and FRMEN (SPIxCON<14>) bits to ‘1’ and SPIFSD (SPIxCON<13>) bit to ‘0’. In this mode, the serial clock will be output continuously at the SCKx pin, regardless of whether the module is transmitting. When the SPIxBUF is written, the SSx pin will be driven high on the next transmit edge of the SCKx clock (active edge depends on the control bit CKP). The SSx pin will be high for one SCKx clock cycle. The module will start transmitting data on the next transmit edge of the SCKx, as shown in Fig. 9-8. The connection of master and slave is shown in Fig. 9-7.

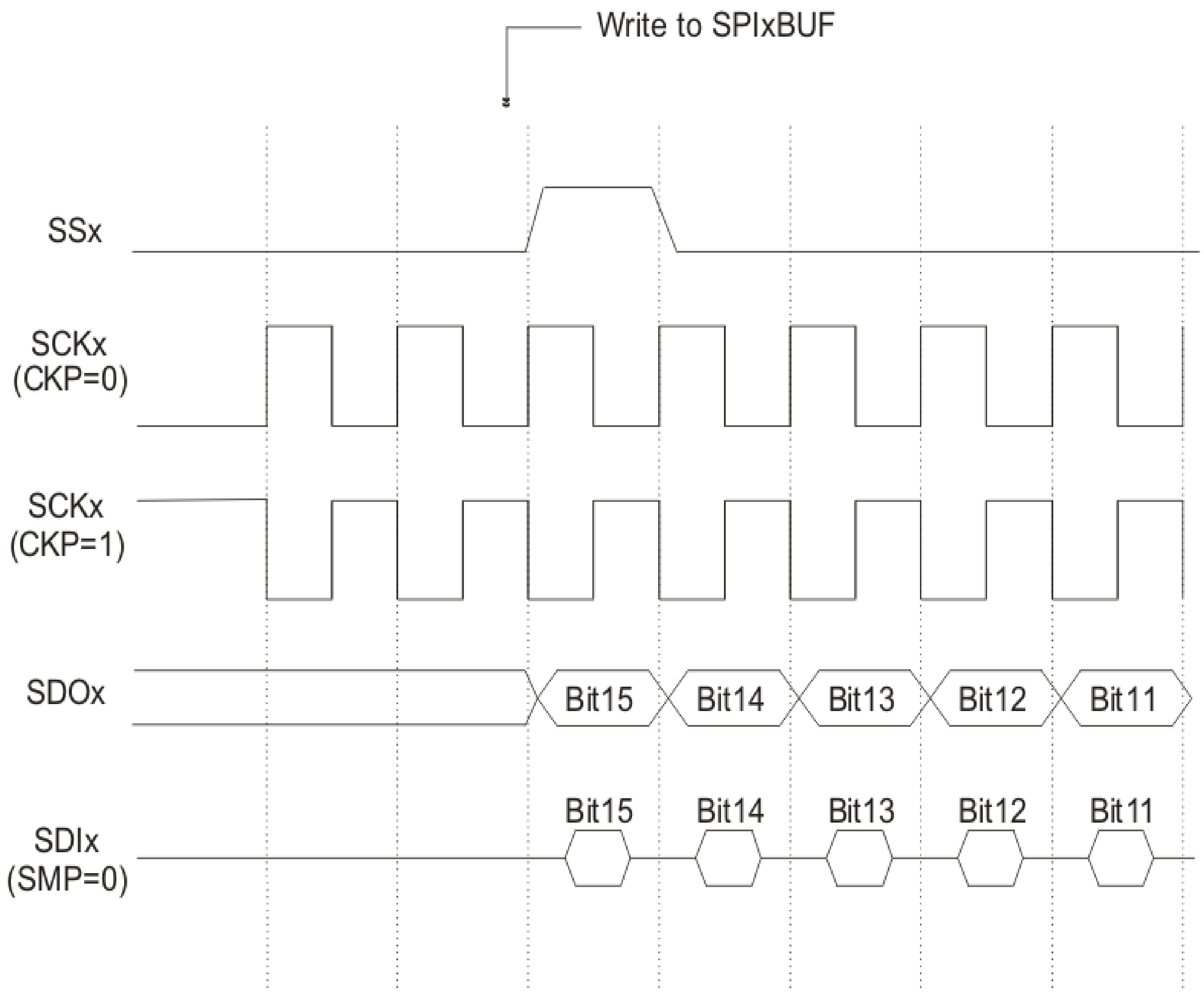


Fig. 9-8 Waveforms of the SPI module in master mode and frame master mode

9.3.2 SPI module in master mode and frame slave mode

This framed SPI mode is enabled by setting the MSTEN (SPIxCON<5>), FRMEN (SPIxCON<14>), and SPIFSD (SPIxCON<13>) bits to ‘1’. The SSx pin is an input, and it is sampled on the sample edge of the SPI clock. When it is sampled high, data will be transmitted on the subsequent transmit edge (controlled by the CKP bit) of the SPI clock. When the transmission is completed, the interrupt flag SPIxIF is generated by the SPIxSR register. The user must make sure that the correct data are loaded into the SPIxBUF for transmission before the signal is received at the SSx pin. Fig. 9-9 shows the waveforms of the SPI module in master mode and frame slave mode. Connection diagram of the module in master mode and frame slave mode is showm in Fig. 9-10.

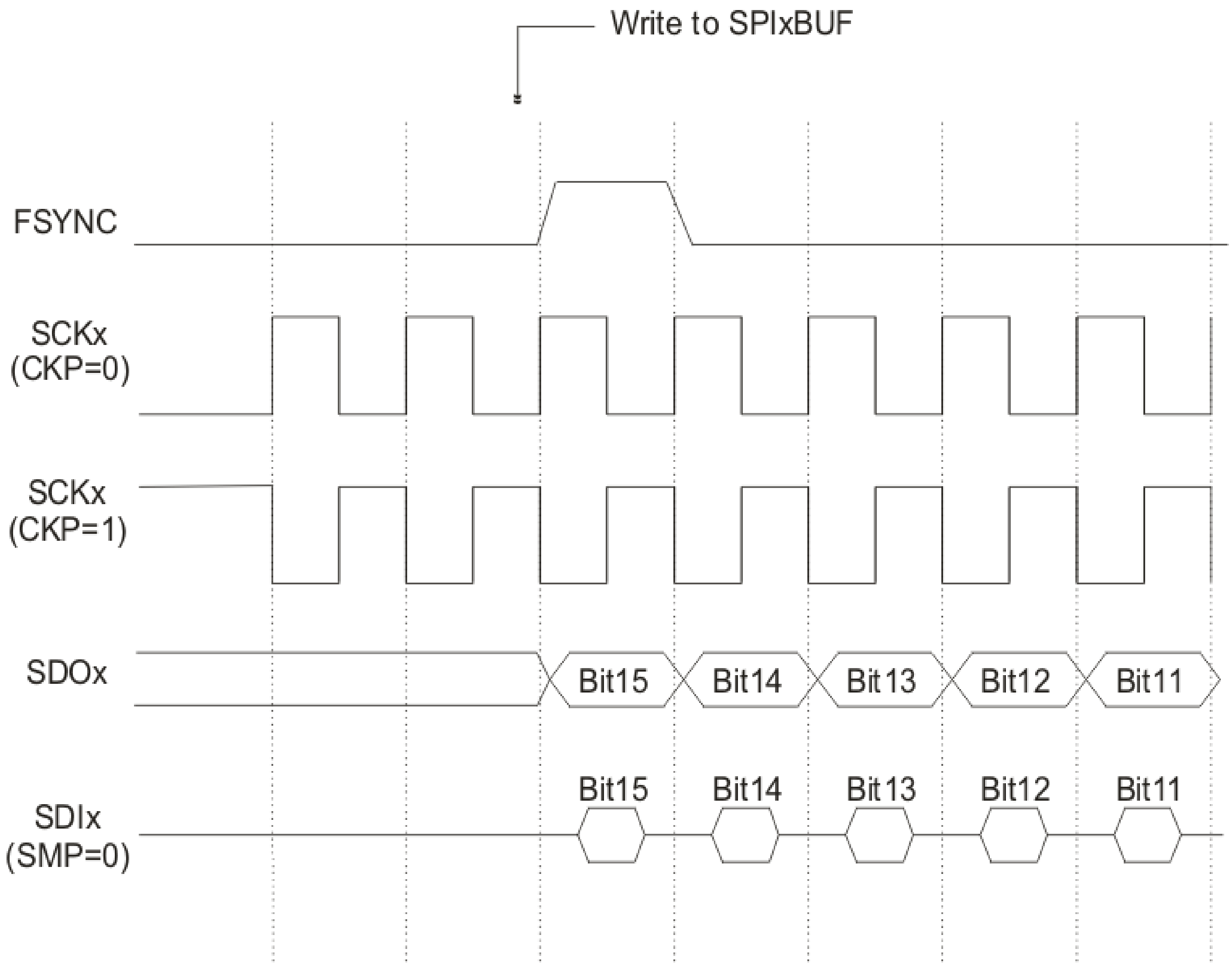


Fig. 9-9 Waveforms of the SPI module in master mode and frame slave mode

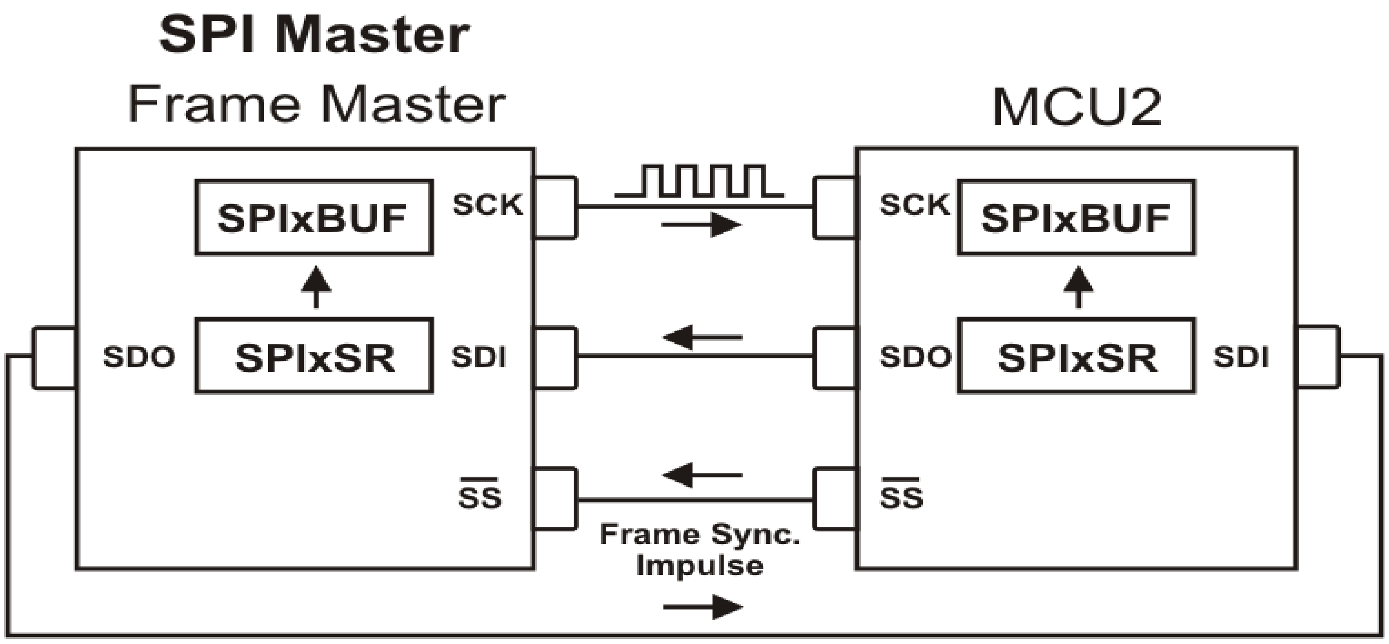


Fig. 9-10 Connection diagram of the module in master mode and frame slave mode

9.3.3 SPI module in slave mode and frame master mode

This framed SPI mode is enabled by setting the MSTEN (SPIxCON<5>) bit to ‘0’, FRMEN (SPIxCON<14>) bit to ‘1’, and SPIFSD (SPIxCON<13>) bit to ‘0’. The input SPI clock will be continuous at the SCKx pin in slave mode. The SSx pin will be an output when the SPIFSD (SPIxCON<13>) bit is cleared. Therefore, when the SPIxBUF is written, the module will drive the SSx pin high on the next transmit edge of the SPI clock. The SSx pin will be driven high for one SPI clock cycle. Data will start transmitting on the next SPI clock falling edge. A connection diagram for this operating mode is shown in Fig. 9-11

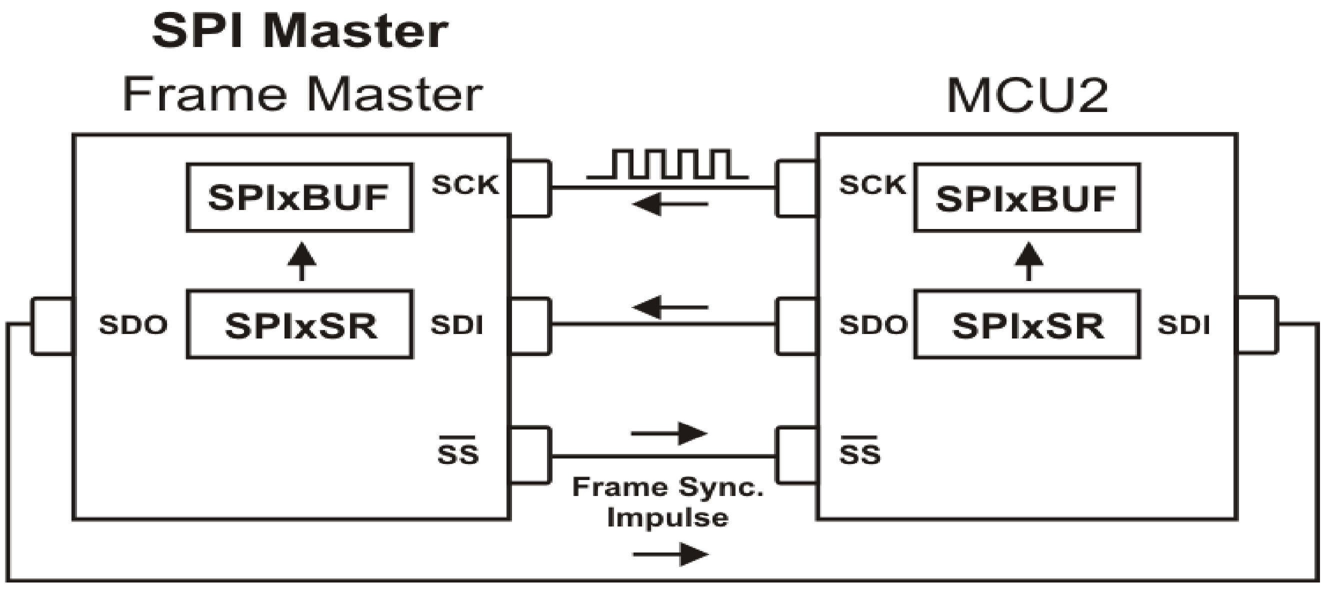


Fig. 9-11 Connection diagram of the module in slave mode and frame master mode

9.3.4 SPI module in slave mode and frame slave mode

This framed SPI mode is enabled by setting the MSTEN (SPIxCON<5>) bit to ‘0’, FRMEN (SPIxCON<14>) bit to ‘1’, and SPIFSD (SPIxCON<13>) bit to ‘1’. Therefore, both the SCKx and SSx pins will be the inputs. The SSx pin will be sampled on the sampling edge of the SPI clock (in the middle of the SPI cycle). When SSx is sampled high, data will be transmitted on the next transmit edge of SCKx (controlled by the CKP bit). A connection diagram for this mode of operation is shown in Fig. 9-12.

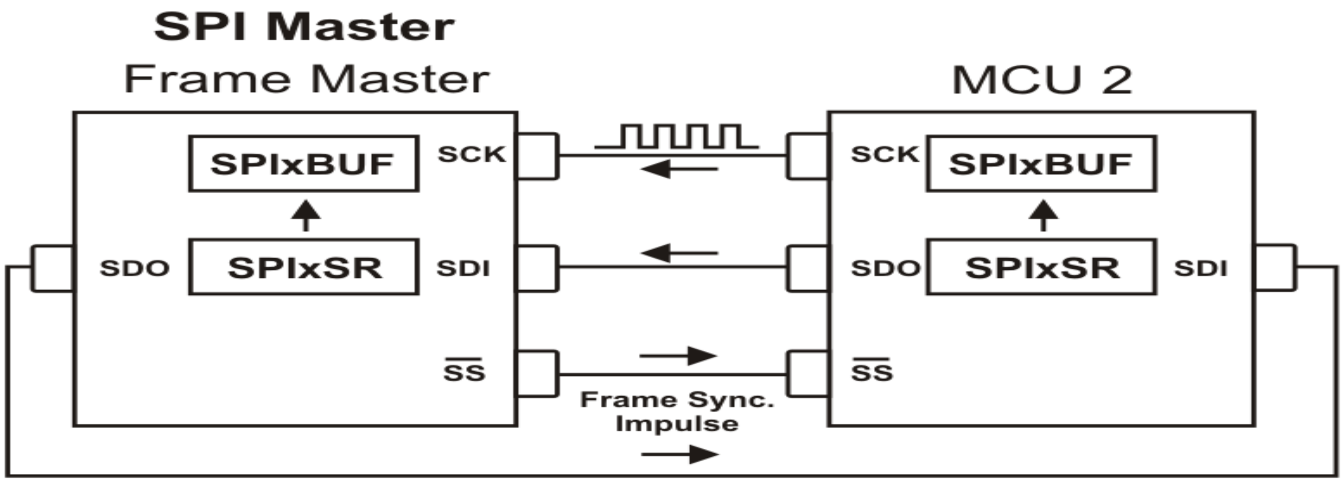


Fig. 9-12 Connection diagram of the module for slave mode and slave frame mode

9.4 SPI master mode clock frequency

In the master mode, the clock provided to the SPI module is the instruction cycle TCY . This clock will then be prescaled by the primary prescaler, specified by PPRE<1:0> (SPIxCON<1:0>), and the secondary prescaler, specified by SPRE<2:0> (SPIxCON<4:2>). The prescaled instruction clock becomes the serial clock and is provided to external devices via the SCKx pin.

Attention!!!
Note that the SCKx signal clock is not free running for normal SPI modes (8-bit or 16-bit). It will only run for 8 or 16 pulses when the SPIxBUF is loaded with data. It will however, be continuous for framed modes.

The SCKx clock frequency as a function of the primary and secondary prescaler settings is calculated by the following equation

$$F_{SCK} = \frac{F_{CY}}{primary\_prescaler \bullet secondary\_prescaler}$$

Examples of the SCKx frequencies as functions of the primary and secondary prescaler settings are shown in Table 9-1.

Fcy = 30 Mhz		**Secondary prescaler settings**				
		1:1	**2:1**	**4:1**	**6:1**	**8:1**
Primary prescaler settings						
	1:1	30 000	15 000	7 500	5 000	3 750
	4:1	7 500	3 750	1 875	1 250	938
	16:1	1 875	938	469	313	234
	64:1	469	234	117	78	59
Fcy = 5 Mhz						
Primary prescaler settings						
	1:1	5 000	2 500	1 250	833	625
	4:1	1 250	625	313	208	156
	16:1	313	156	78	52	39
	64:1	78	39	20	13	10

Table 9-1 SCKx frequencies

NOTE: SCKx clock frequencies shown in kHz. All frequencies are not supported; electrical characteristics of individual microcontrollers of dsPIC30F family should be consulted.

Example:

The example shows the use of the specialized SPI library of the mikroC compiler for dsPIC microcontrollers which allows an easy initialization of the SPI module and write or read data from the transmit/receive buffer of the SPI module. The example shows the method of connecting the SPI2 module to the serial digital-to-analogue converter (DAC) MCP4921.

```
const char CS_PIN = 0;

unsigned int value;



void InitMain() {

  ADPCFG = 0xFFFF;                        // Set AN pins as digital



  Spi_Init();                             // Initialize SPI module



  TRISF.CS_PIN = 0;                       // Set CS pin as output

}//~



// DAC increments (0..4095) --> output voltage (0..Vref)

void DAC_Output(unsigned int valueDAC) {

 char temp;



  PORTF.CS_PIN = 0;                       // Select DAC module



  // Send 2 bytes of valueDAC variable
  temp = (valueDAC >> 8) & 0x0F;          // Prepare hi-byte for transfer

                                          // It's a 12-bit number, so only

                                          // lower nibble of high byte is
used

  temp |= 0x30;                           // Set MCP4921 control bits

  Spi_Write(temp);                        // Send data via SPI



  temp = valueDAC;                        // Prepare lo-byte for transfer

  Spi_Write(temp);                        // Send data via SPI



  PORTF.CS_PIN = 1;                       // Deselect DAC module

}//~



void main() {

  InitMain();



  value = 2047;                           // When program starts, DAC gives

                                          // the output in the mid-range



  while (1) {                             // Main loop

    DAC_Output(value++);

    if (value > 4095)

      value = 0;
    Delay_ms(10);


  }


}//~!
```

Function Spi2_Init initializes the SPI module in master mode 8-bit formatted, without the SS2 pin, sampling in the middle of the SPI cycle, prescale 1:8, and the clock FCY:1 low while waiting. If another format or communication speed is desired, the function Spi2_Init_Advanced instead of Spi2_Init should be used. The function Spi2_Data_Ready reads the value of the status bit SPIRBF (SPI2STAT<0>) and checks if there are data loaded into the receive buffer register. The function Spi2_Write transmits data via the SPI module. Data reception is performed by the function Spi2_Read. The SPI moduled is disabled by the function Spi2_Stop (clearing the control bit SPIEN), and the SPI module enable (setting the control bit SPIEN) is performed automatically by the function Spi2_Init or Spi2_Init_Advanced.

9.5 SPI module operation in SLEEP and IDLE modes

9.5.1 SPI module operation in SLEEP mode

When the device enters SLEEP mode, the system clock is disabled. If the SPI module is in master mode when the microcontroller enters SLEEP mode, the SPI clock is also disabled. If the SPIx module enters SLEEP mode in the middle of a transmission/reception, then the transmission/reception is aborted. Since there is no automatic way to prevent an entry into SLEEP mode, the user software must synchronize entry into SLEEP with SPI module operation. The transmitter or receiver does not continue with partially completed transmission at wake-up.

Since the clock pulses at SCKx are externally provided for slave mode, the module will continue to function in SLEEP mode. It will complete any transaction during the transition into SLEEP. On completion of a transaction, the SPIRBF status flag for interrupt request of the SPI module is set. If the SPI interrupts are enabled (SPIxIE=1), the device will wake-up from SLEEP. If the SPI interrupt periority level is greater than the present CPU priority level, code execution will resume at the SPIx interrupt vector location. Otherwise, code execution will continue with the instruction that previously invoked SLEEP mode. Entering or waking-up from SLEEP mode does not influence the operation of the SPI module in slave mode nor does it change the contents of the SPI module registers.

9.5.2 SPI module operation in IDLE mode

When the device enters IDLE mode, the system clock sources remain functional. The SPISIDL bit (SPIxSTAT<13>) selects whether the module will stop or continue functioning on IDLE.

If SPISID (SPIxSTAT<13>) bit is cleared, the SPI module will continue operation in IDLE mode.

If SPISID (SPIxSTAT<13>) bit is set, the SPI module will stop communication on entreing IDLE mode. It will operate in the same manner as it does in SLEEP mode.

Table 9-2 describes pins of the SPI module depending on the operational mode. At the end of this section a description of the UART registers for dsPIC30F is presented.

Name	Type	Description
SCKx	Input / Output	SPI clock signal output or input
SDIx	Input	SPI data reception input pin
SDOx	Output	SPI data transmisson output pin SPI slave device selection control pin Write/Read enable pin in slave mode if the control bit SSEN (SPIxCOPN<7>) is set
SSx	Input / Output	Used as RAM synchronization pin when the control bits FRMEN and SPIFSD (SPIxCON<13:14>) are set to the value 10 or 11

Table 9-2 Description of SPI module pins

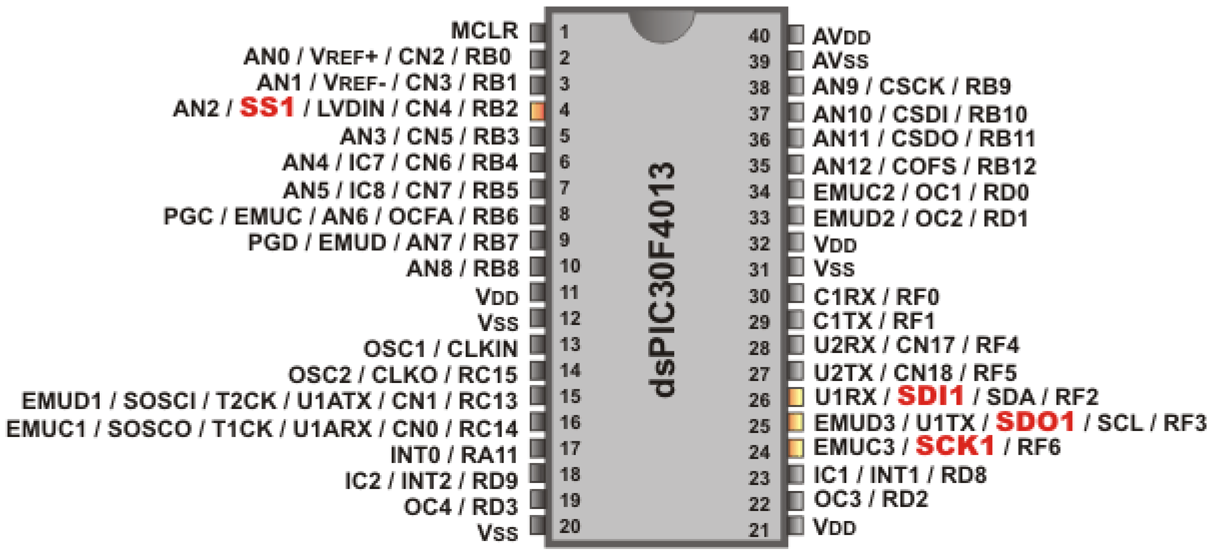


Fig. 9-13a Pinout of dsPIC30F4013

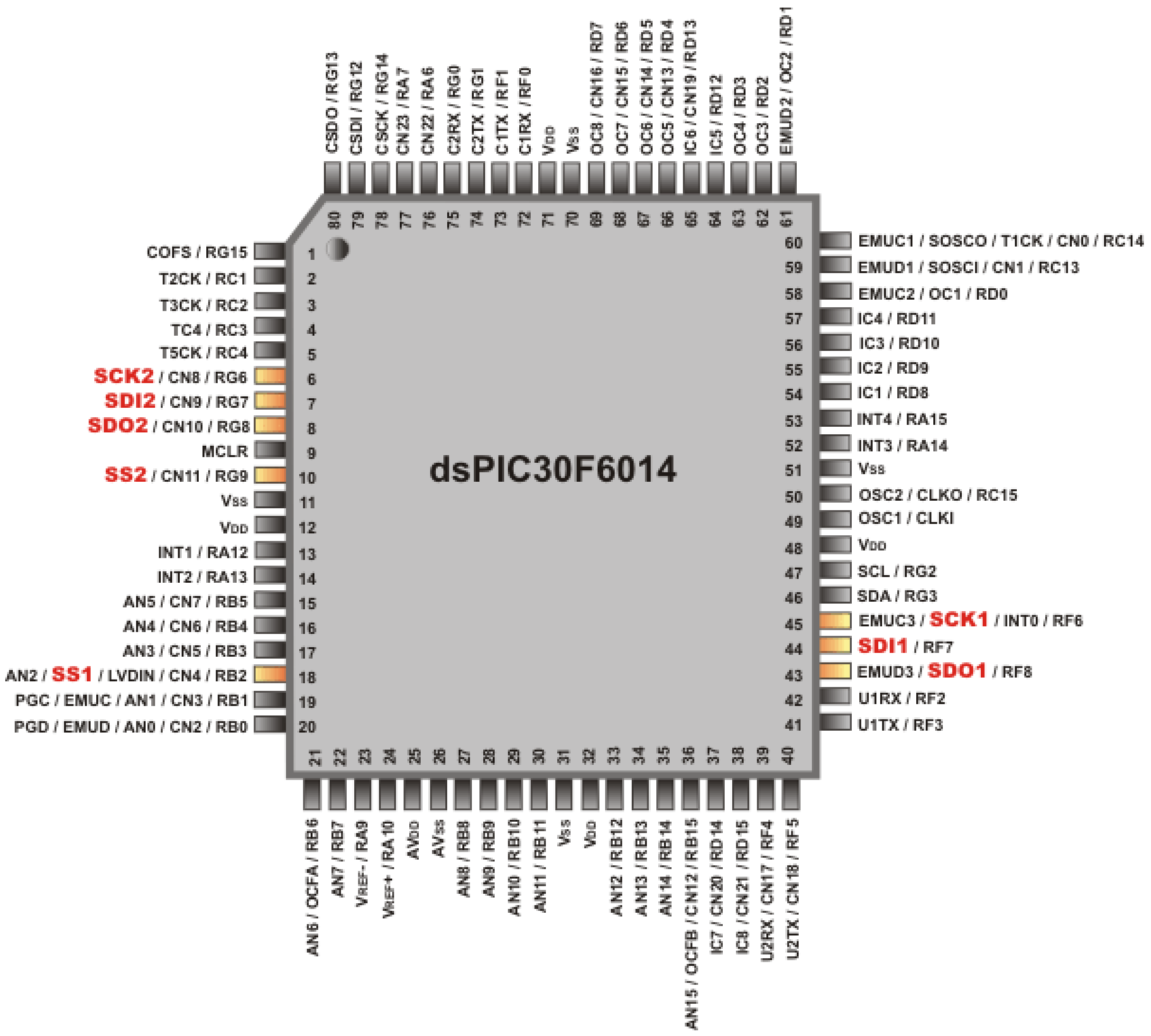


Fig. 9-13b Pinout of dsPIC30F6014A

A brief description follows of the SPI module registers of a dsPIC30F4013 device.

name	ADR	15	14	13	12-7	6	5	4	3	2	1	0	Reset State
SPI1STAT	0x0220	SPIEN	-	SPISIDL	-	SPIROV	-	-	-	-	SPITBF	SPIRBF	0x0000

Table 9-3 Status and control register SPI1STAT

SPIEN – SPI enable bit (SPIEN=0 disables module, SPIEN=1 enables module)
SPISIDL – Stop in IDLE mode bit (SPISIDL=0 continue operation, SPISIDL=1 discontinue operation)
SPIROV – Receive overflow flag bit
SPITBF – SPI transmit buffer full status bit (SPITBF=0 transmit started, SPIxTBF empty,
SPITBF=1 transmit not yet started, SPIxTBF is full)
SPIRBF – SPI receive buffer full status bit (SPIRBF=0 receive is not complete
SPIxRXB is empty, SPIRBF=1 receive complete SPIxRXB is full)

name	ADR	15	14	13	12	11	10	9	8	7	6	5
SPI1CON	0x0222	-	FRMEN	SPIFSD	-	DISSDO	MODE16	SMP	CKE	SSEN	CKP	MSTEN

Table 9-4a Control register SPI1CON

4	3	2	1	0	Reset State

SPRE<2:0>	PPRE<1:0>	0x0000

Table 9-4b continued

```
FRMEN – Framed SPI support bit
SPIFSD – Frame sync pulse direction control on SSx pin bit
(SPIFSD=0 frame sync pulse output (master), SPIFSD=1 frame sync pulse input
(slave))
DISSDO – Disable SDOx pin bit
MODE16 – Word/byte communication select bit (MODE16=0 8-bit mode, MODE16=1
16-bit mode)
SMP – SPI data input sample phase bit (Master mode: SMP=0 input data
sampled at middle of
data output time, SMP=1 input data sampled at end of data output time;
Slave mode: SMP must be cleared)

CKE – SPI clock edge select bit (CKE=0 serial output data changes on
transition from
IDLE clock state to active clock state, CKE=1 serial output data changes on
transition from
active clock state to IDLE clock state)

SSEN – Slave select enable bit
(SSEN=0 SS1 pin not used by module, SSEN=1 SS1 pin used for slave mode)

CKP – Clock polarity select bit (CKO=0 IDLE state for clock is a low level,
active state
is a high level, CKO=1 IDLE state for clock is a high level, active state
is a low level)

MSTEN – Master mode enable bit (MSTEN=0 slave mode, MSTEN=1 master mode)
SPRE<2:0> - Secondary prescale (master mode) bits
     000 – secondary prescale 8:1
     001 – secondary prescale 7:1
     ...
     110 – secondary prescale 2:1
     111 – secondary prescale 1:1
PPRE<1:9> - Primary prescale (master mode) bits
     00 – primary prescale 64:1
     01  - primary prescale 16:1
     10 – primary prescale 4:1
     11 – primary prescale 1:1
```

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
SPI1BUF	0x0224	Transmit/Receive buffer (shared by SPI1TXB and SPI1RXB registers)																0x0000

Table 9-5 SPI1BUF register

Chapter10: UART Module

Introduction

The Universal Asynchronous Receiver Transmitter (UART) module is the basic serial I/O module available in the dsPIC30F device family. The UART is a full-duplex asynchronous system that can communicate with peripheral devices, such as personal computers, RS-232, and RS-485 interfaces.

The primary features of the UART module are:

- Full-duplex 8- or 9-bit data transmission through the UxTX and UxRX pins,
- For 8-bit data even, odd, or no parity options,
- One or two STOP bits,
- Fully integrated Baud-rate generator with 16-bit prescaler,
- 4-deep First-In-First-Out (FIFO) transmit data buffer,
- 4-deep FIFO receive data buffer,
- Parity, framing and buffer overrun error detection,
- Support for 9-bit mode with address detect (9th bit=1),
- Transmit and receive interrupts,
- Loop-back mode for diagnostic support.

Each dsPIC30F device variant may have one or more UART modules (e.g. dsPIC30F4013 has 2 UART modules).

Fig. 10-1 shows a simplified block diagram of the UART module. The UART module consists of the three key hardware elements: Baud-rate generator, asynchronous transmitter, and asynchronous receiver.

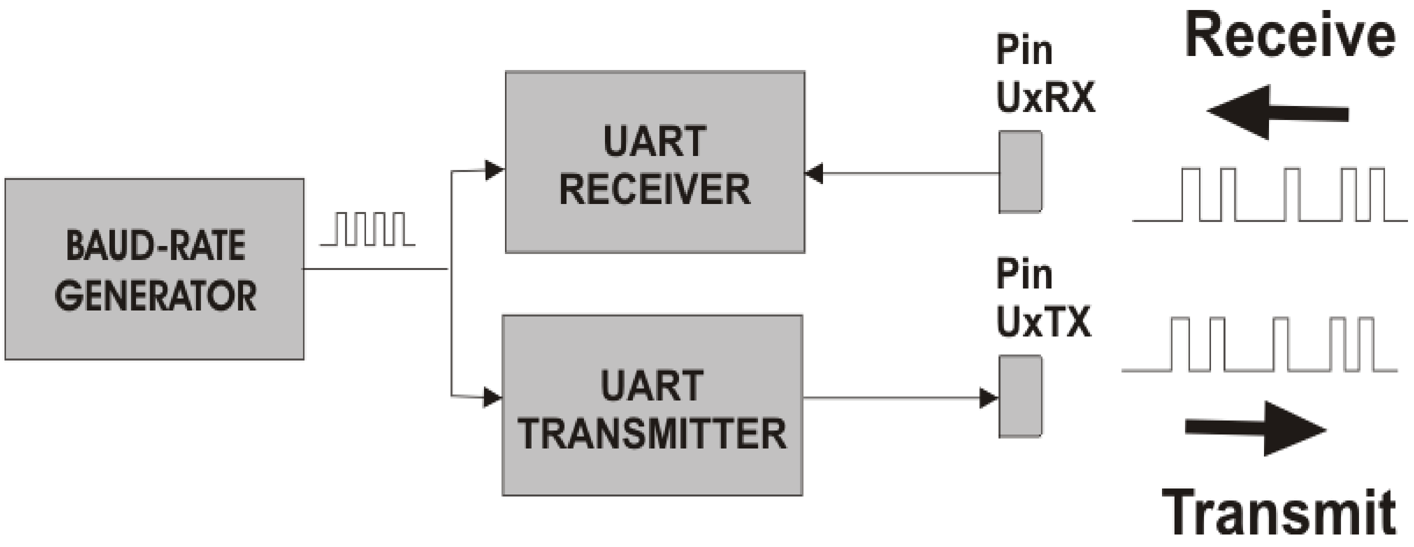


Fig. 10-1 UART simplified block diagram

10.1 Baud-rate generator BRG

The UART module includes a dedicated 16-bit baud rate generator with a prescaler. The UxBRG register controls the period of a free-running 16-bit timer:

$$Baud\_rate = \frac{F_{CY}}{16 \cdot (UxBRG + 1)}$$

the value of the UxBRG register for a specified baud rate is given by

$$UxBRG = \frac{F_{CY}}{16 \cdot Baud\_rate} - 1$$

Example:
If FCY=8MHz, and the desired baud rate is 9600, by applying the above expression one obtains the value UxBRG=(8•106/(16•9600))-1=51.083. After truncation to the integer value UxBRG=51. The truncation introduced a deviation from the desired baud rate. The new baud rate for UxBRG=51 is 9615.384, i.e. the deviation is 0.16%.

The baud rate and the deviation from a desired baud rate are strongly dependent on the basic clock of the device and the instruction cycle FCY. UART baud rates, baud rate deviations, and UxBRG register values as functions of the device instruction cycle frequencies are shown in Table 10-1.

Baud rate [Kbps]	**Fcy=30MHz**			**Fcy=25MHz**			**Fcy=20MHz**			**Fcy=16MHz**		
	KBAUD	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**
0.3	0.3	0.0	6249	0.3	+0.01	5207	0.3	0.0	4166	0.3	+0.01	3332
1.2	1.1996	0.0	1562	1.2001	+0.01	1301	1.1996	0.0	1041	1.2005	+0.04	832
2.4	2.4008	0.0	780	2.4002	+0.01	650	2.3992	0.0	520	2.3981	-0.08	416
9.6	9.6154	+0.2	194	9.5859	-0.15	162	9.6154	+0.2	129	9.6154	+0.16	103
19.2	19.1327	-0.4	97	19.2901	+0.47	80	19.2308	+0.2	64	19.2308	+0.16	51
38.4	38.2653	-0.4	48	38.1098	-0.76	40	37.8788	-1.4	32	38.4615	+0.16	25
56	56.8182	+1.5	32	55.8036	-0.35	27	56.8182	+1.5	21	55.5556	-0.79	17

115	117.1875	+1.9	15	111.607	-2.95	13	113.6364	-1.2	10	111.1111	-3.38	8
250							250	0.0	4	250	0.0	3
500										500	0.0	1
MIN	0.0286	0.0	65535	0.0238	0.0	65535	0.019	0.0	65535	0.015	0.0	65535
MAX	1875	0.0	0	1562.5	0.0	0	1250	0.0	0	1000	0.0	0

Baud rate [Kbps]	**Fcy=12MHz**			**Fcy=10MHz**			**Fcy=8MHz**			**Fcy=7.68MHz**		
	KBAUD	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**
0.3	0.3	0.0	2499	0.3	0.0	2082	0.2999	-0.02	1666	0.3	0.0	1599
1.2	1.2	0.0	624	1.1996	0.0	520	1.199	-0.08	416	1.2	0.0	399
2.4	2.3962	-0.2	312	2.4038	+0.2	259	2.4038	+0.16	207	2.4	0.0	199
9.6	9.6154	-0.2	77	9.6154	+0.2	64	9.6154	+0.16	51	9.6	0.0	49
19.2	19.2308	+0.2	38	18.9394	-1.4	32	19.2308	+0.16	25	19.2	0.0	24
38.4	37.5	+0.2	18	39.0625	+1.7	15	38.4615	+0.16	12			
56	57.6923	-2.3	12	56.8182	+1.5	10	55.5556	-0.79	8			
115			6									
250	250	0.0	2				250	0.0	1			
500							500	0.0	0			
MIN	0.011	0.0	65535	0.010	0.0	65535	0.008	0.0	65535	0.007	0.0	65535
MAX	750	0.0	0	625	0.0	0	500	0.0	0	480	0.0	0

Baud rate [Kbps]	**Fcy=5MHz**			**Fcy=4MHz**			**Fcy=3.072MHz**			**Fcy=1.8432MHz**		
	KBAUD	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**	**KBAUD**	**Error [%]**	**BRG**
0.3	0.2999	0.0	1041	0.3001	0.0	832	0.3	0.0	639	0.3	0.0	383
1.2	1.2019	+0.2	259	1.2019	+0.2	207	1.2	0.0	159	1.2	0.0	95
2.4	2.4038	+0.2	129	2.4038	+0.2	103	2.4	0.0	79	2.4	0.0	47
9.6	9.4697	-1.4	32	9.6154	+0.2	25	9.6	0.0	19	9.6	0.0	11
19.2	19.5313	+1.7	15	19.2308	+0.2	12	19.2	0.0	9	19.2	0.0	5
38.4	39.0625	+1.7	7				38.4	0.0	4	38.4	0.0	2
56												
115												
250												
500												
MIN	0.005	0.0	65535	0.004	0.0	65535	0.003	0.0	65535	0.002	0.0	65535
MAX	312.5	0.0	0	250	0.0	0	192	0.0	0	115.2	0.0	0

Table 10-1 Baud rates, baud rate deviations, and UxBRG register values as functions of the device instruction cycle frequencies

Attention!
The minimum baud rate is FCY/(16•65535) for UxBRG=0xFFFF. The maximum baud rate is FCY/16 for UxBRG=0. Writing a new value to the UxBRG register causes the BRG timer to be cleared. This ensures that the BRG immediately starts generating the new baud rate.

10.2 UART configuration

The UART uses standard non-return-to-zero (NRZ) format consisting of one START bit, eight or nine data bits, and one or two STOP bits. To increase data transmission noise immunity, parity is supported by the hardware. The UART may be configured by the user as even (logic zero or logic one is added to make the total number of units in a message even), odd or no parity.

The most common data format is 8 bits, no parity and one STOP bit (denoted as 8, N, 1), which is the default setting.

The number of data bits and the parity are specified in the PDSEL<1:0> (UxMODE<2:1>) and the number of STOP bits in the STSEL (UxMODE<0>) control bit.

The UART baud rate is configured by setting the UxBRG register.

Note:
The UART module transmits and receives the LSb first. The UART's transmitter and receiver are functionally independent, but use the same data format and baud rate.

The UART module is enabled by setting the UARTEN (UxMODE<15>) control bit. The transmit mode of the UART module is enabled by the UTXEN (UxSTA<10>) bit. Once enabled, the UxTX and UxRX pins are configured as an output and an input, respectively, overriding the TRIS and PORT register bit settings for the corresponding I/O port pins. The UxTX is at logic '1' when no transmission is taking place. The UART module is disabled by clearing the UARTEN (UxMODE<15>) control bit. This is the default state after any RESET. If the UART module is disabled, all UART pins operate as port pins under the control of their corresponding PORT and TRIS bits. Disabling the UART module resets the FIFO buffers to empty states and the baud rate counter is reset.

When the UARTEN (UxMODE<15>) bit is cleared, all error and status flags associated with the UART module: URXDA, OERR, FERR, PERR, UTXEN, UTXBRK, and UTXBF are cleared, whereas the RIDLE and TRMT bits are set to '1'. Other control bits are of no influence as long as UARTEN= 0. Clearing the UARTEN bit while the UART is active will abort all pending transmissions and receptions and reset the module. Re-enabling the UART will restart the UART in the same configuration.

Some dsPIC30F devices have an alternative set of UART transmit and receive pins that can be used for serial communication. These are very useful when the primary UART pins are shared by other preipherals. The alternate I/O pins are enabled by setting the ALTIO control bit. If ALTIO=1, the UxATX and UxARX pins are used by the UART module, instead of UxTX and UxRX.

10.3 UART transmitter

The UART transmitter block diagram is shown in Fig. 10-2. The heart of the transmitter is the transmit shift register UxTSR where parallel data (9-bit word) are converted to serial data sequences. The shift register obtains its data from the transmit FIFO buffer, TxTXREG. The write-only UxTXREG register is loaded with data by the user software. The UxTXREG is not loaded until the STOP bit has been transmitted from the previous load. As soos as the STOP bit is transmitted, the UxTSR is loaded with new data from the UxTXREG register (if available).

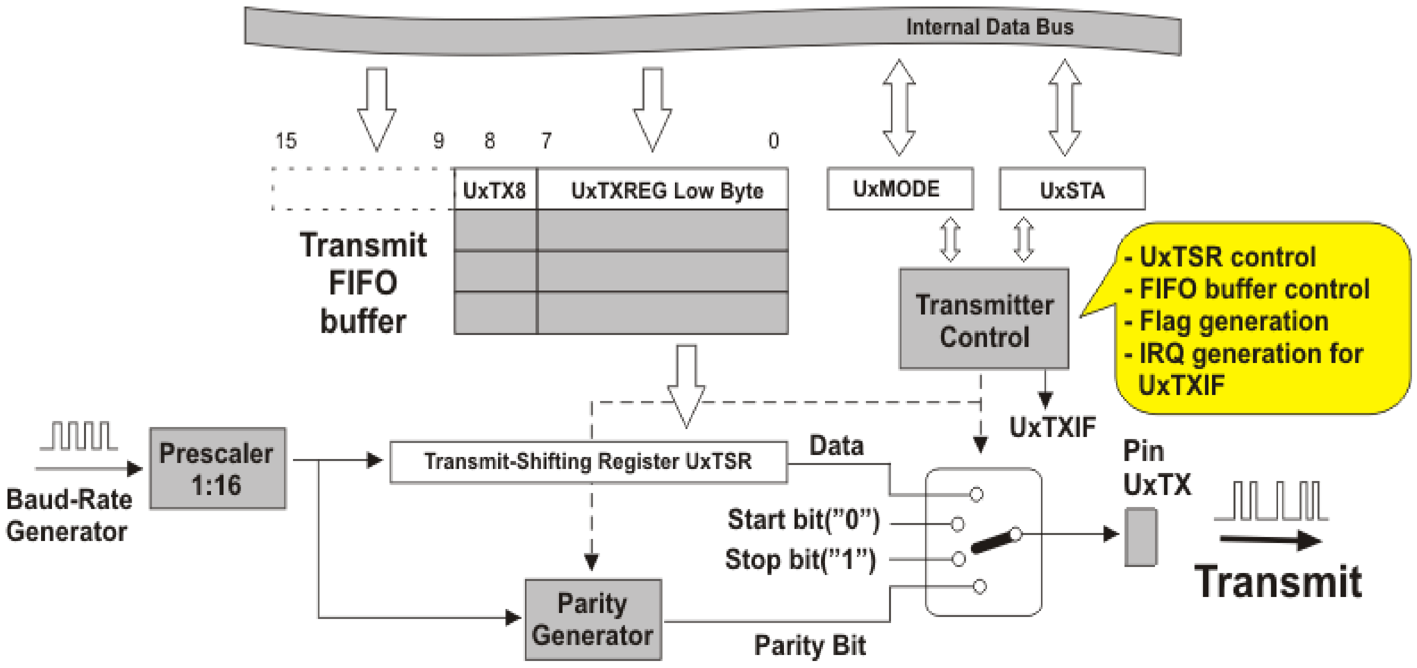


Fig. 10-2 UART transmitter functional block diagram

Attention!
The registers UxTXREG and UxTSR are 9-bit wide, i.e. data write to the transmit FIFO buffer through the register UxTXREG is done in bytes (8-bit). The UxTSR register is not mapped in data memory, thus it is not available to the user.

The UART transmission is enabled by setting the UTXEN enable bit (UxSTA<10>). The actual transmission will not occur until the UxTXREG has been loaded with data and the baud rate generator has produced a shift clock, in accordance with the value in the register UxBRG. The transmission can also be started by first loading the UxTXREG register and the setting the UTXEN enable bit.

Clearing the UTXEN bit during a transmission will cause the transmission to be aborted and will reset the transmitter. As a result, the UxTX pin will revert to a high-impedance state.

In order to select 9-bit transmission, the PDSEL<1:0> bits (UxMODE<2:1>) should be set to '11' and the ninth bit should be written to the 9th location of the UxTxREG register (UxTXREG<8>). A word (16-bit) should be written to UxTXREG so that all nine bits are written at the same time.

\

NOTE: There is no parity in the case of 9-bit data transmission.

The data transmit FIFO buffer consists of four 9-bit wide memory locations. The UxTXREG register provides user access to the next available buffer location. The user may write up to 4 words in the buffer. Once the first location in the transmit FIFO buffer is loaded to the shift register UxTSR, that location becomes available for new data to be written and the next location is sourced to the UxTSR register. The UTXBF status bit is set whenever the buffer is full (all four locations). If a user attempts to write to a full buffer, the new data will not be accepted into the FIFO.

Attention!
The FIFO is reset during any device RESET, but is not affected when the device enters SLEEP or IDLE mode or wakes up from SLEEP or IDLE mode.

10.3.1 Transmit interrupt

The transmit interrupt flag (UxTXIF) is located in the corresponding interrupt flag status (IFS) register. The UTIXSEL control bit (UxSTA<15>) determines when the UART will generate a transmit interrupt.

- If UTXISEL = 0, an interrupt is generated when a word (8- or 9-bit) is transferred from the transmit FIFO buffer to the transmit shift register (UxTSR). This implies that the transmit buffer has at least one empty word. An iterrupt is generated very often. If the UART module is to operate in this mode, it is required that the corresponding interrupt service routine is very quick, i.e. it should be completed before the transmission of the next word.

- If UTXISEL = 1, an interrupt is generated when a word is transferred from the transmit FIFO buffer to the transmit shift register (UxTSR) and the transmit buffer is empty. Since an interrupt is generated only after all 4 words have been transmitted, this 'block transmit' mode is useful if the interrupt service routine cannot be performed very quickly.

Attention!
When the UTXEN enable bit is set, an interrupt request will be generated. The interrupt request should be reset by the user software. If UTXISEL = 0, an interrupt request will be generated on setting the UTXEN since there is at least one empty location in the transmit FIFO buffer; if UTXISEL = 1, an interrupt request will be generated on setting the UTXEN since the transmit FIFO buffer is empty.

While the UxTXIF flag bit indicates the status of the UxTXREG register, the TRMT (UxSTA<8>) status bit shows the status of the UxTSR register. The TRMT status bit is a read only bit, which is set when the UxTSR register is empty. No interrupt logic is tied to this bit, so the user has to poll this bit in order to determine if the UxTSR register is empty.

10.3.2 Setup for UART transmit

When setting up a transmission, the following steps should be undertaken:

1. Initialize the UxBRG register for the appropriate baud rate.
2. Set the number of data bits, number of STOP bits, and parity selection by writing to the PDSEL<1:0> (UxMODE<2:1>) and STSEL (UxMODE<0>) bits.
3. If transmit interrupts are desired, set the UxTXIE control bit in the corresponding interrupt enable control register (IEC). Specify the interrupt priority usin the UxTXIP<2:0> control bits in the corresponding interrupt priority control register (IPC). Select the transmit interrupt mode by writing the UTXISEL (UxMODE<15>) bit.
4. Enable the UART module by setting the UARTEN (UxMODE<15>) bit.

5. Enable the transmission by setting the UTXEN (UxSTA<10>) bit. This will also set the transmit interrupt flag UxTXIF bit. During the initialization, the interrupt request of the UART module transmitter UxTXIF bit should be cleared. Also in the interrupt service routine the interrupt request UxTXIF should be cleared.
6. Finally, load data to the transmit FIFO buffer by writing to the UxTXREG register. If 8-bit transmission is used, load a byte (8-bits). If 9-bit transmission has been selected, load a word (9-bits, higher bits are ignored).

Fig. 10-3 shows the waveforms of an example of serial data transmission (8-bit or 9-bit) by the UART module. Fig. 10-4 shows the waveforms of an example of serial data transmission for a sequence of two bytes.

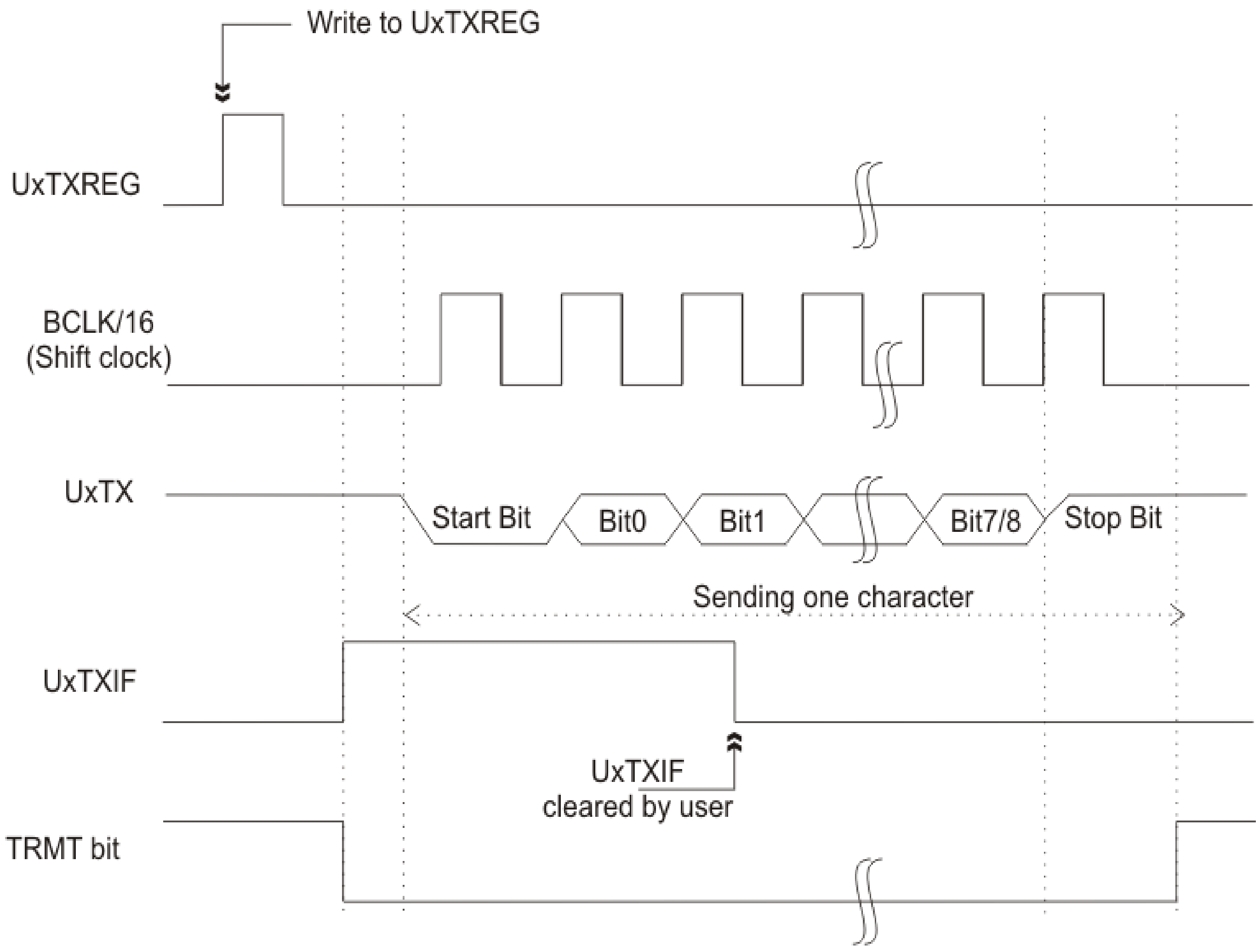


Fig. 10-3 Waveforms of serial data transmission (8 or 9-bits)

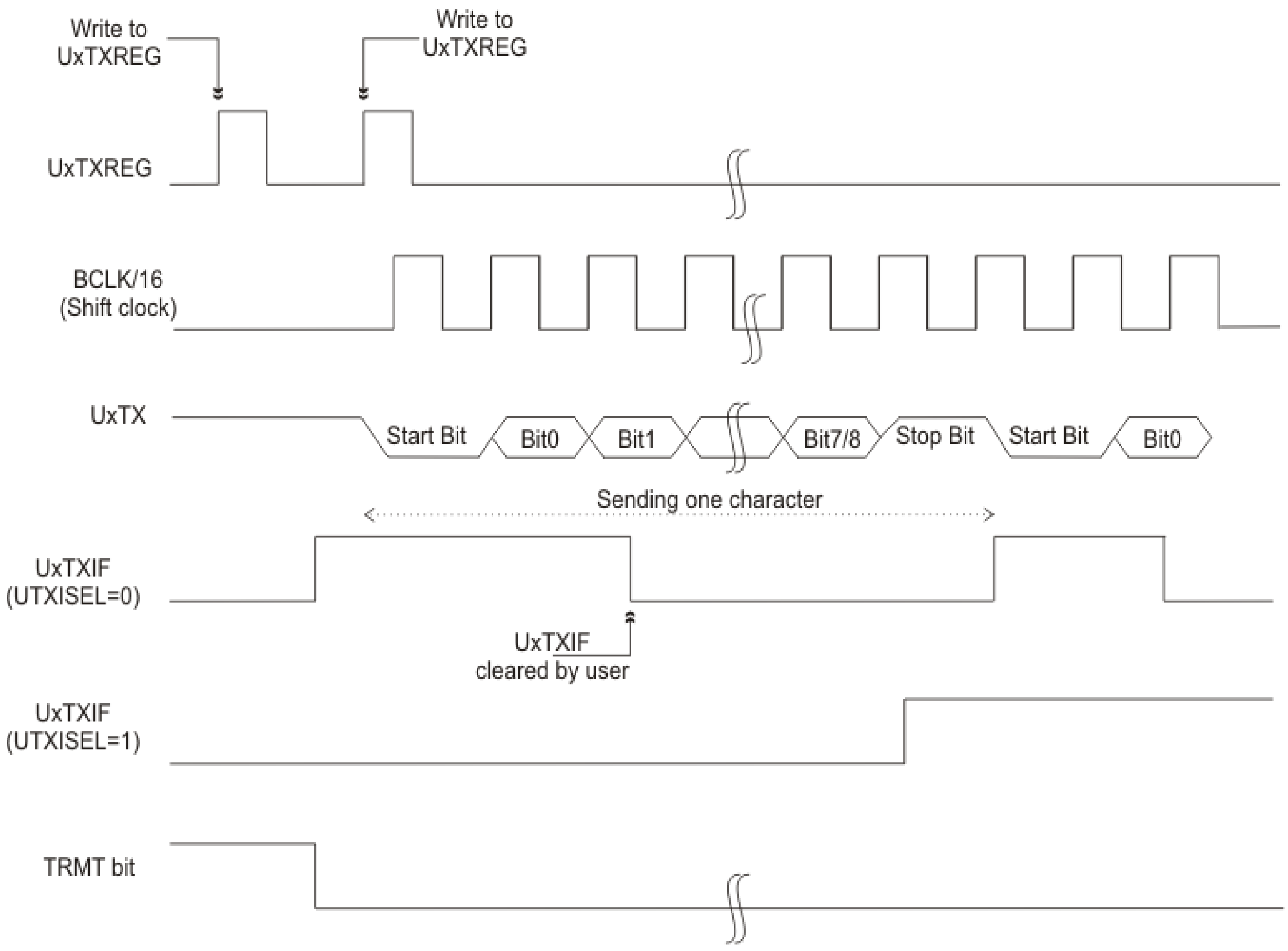


Fig. 10-4 Waveforms of serial data transmission (sequence of two bytes)

For sending a break character (symbol), the UTXBRK (UxSTA<11>) bit must be set by software. Setting this bit forces the output pin UxTX to logic zero. Setting the UTXBRK bit will override any other transmitter activity. The user should wait for the transmitter to complete the current transmission before setting UTXBRK.

To send a break character, the UTXBRK bit must be set by software and remain set for a minimum of 13 baud clocks.The baud clock periods are timed in software. The UTXBRK bit is then cleard by software to generate the STOP bit (one or two, defined by the configuration). The user must wait one or two baud clocks to ensure a valid STOP bit(s) before loading data to the FIFO buffer via the UxTXREG register.

Attention!
Sending a break character does not generate a transmitter interrupt.

10.4 UART receiver

The UART receiver functional block diagram is shown in Fig. 10-5. The heart of the receiver is the receive shift register UxRSR where a serial sequence is converted to a parallel word (9-bit word). After sampling the UxRX pin for the STOP bit, the received data in UxRSR are transferred to the receive FIFO buffer, if it is empty.

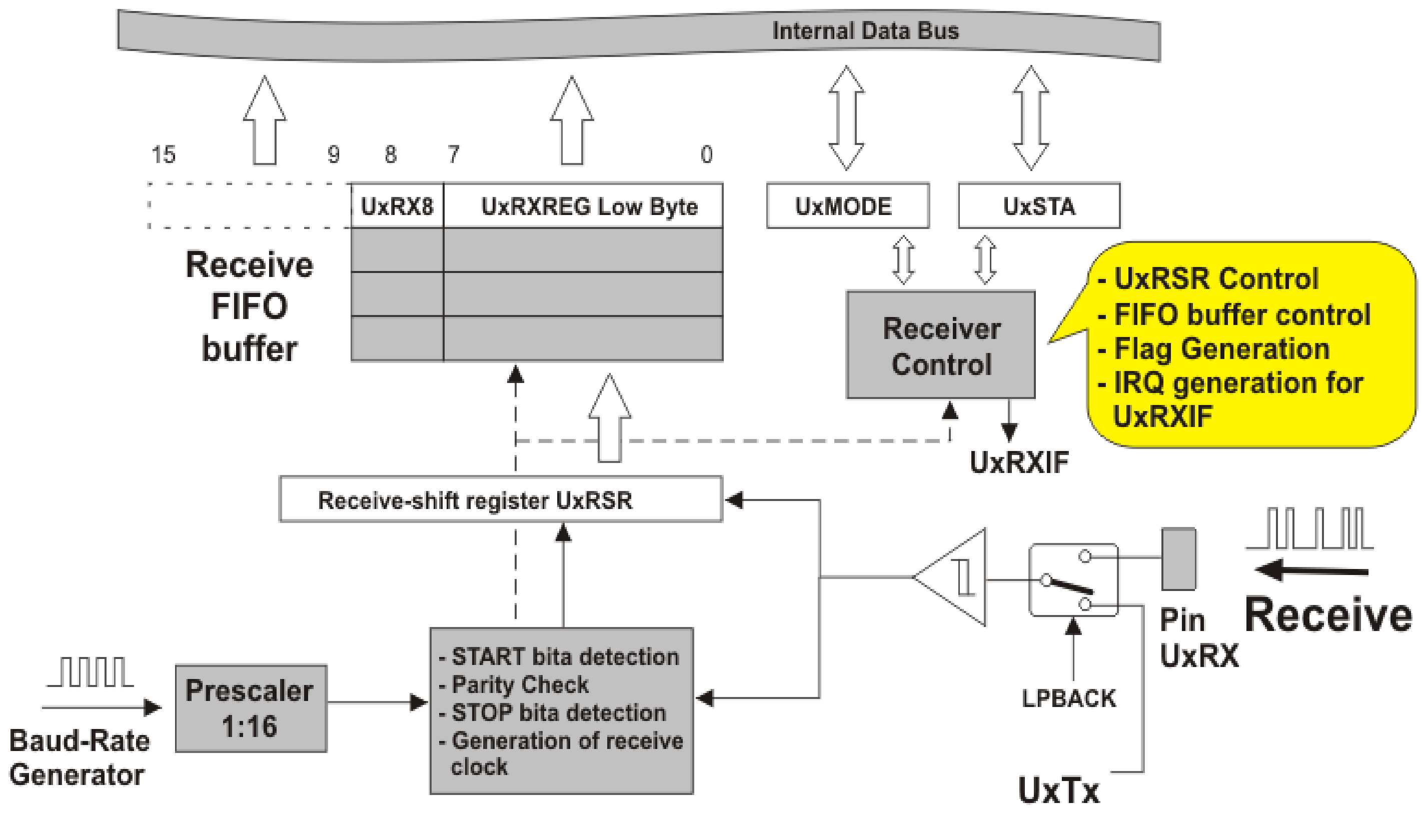


Fig. 10-5 UART receiver functional block diagram.

The data on the UxRX pin are sampled three times by a majority detect circuit to determine if a high or a low level is present at the UxRX pin.

Attention!
The UxRSR register is nor mapped in data memory, so it is not available to the user.

The FIFO receive data buffer consists of four 9-bit wide memory locations. The access to the contents of the receive FIFO buffer is via the UxRXREG read-only register. It is possible for 4 words of data to be received and transferred to the FIFO buffer and a fifth word to begin shifting data to the UxRSR register before a buffer overrun occurs. When the FIFO is full (four characters) and a fifth character is fully received into the UxRSR register, the overrun error bit OERR (UxSTA<1>) will be set. The word in UxRSR will be kept, but further transfers to the receive FIFO are inhibited as long as the OERR bit is set. The user must clear the OERR bit in software to allow further data to be received. Clearing the OERR bit, clears the receive FIFO buffer.

Attention!
The data in the receive FIFO should be read prior to clearing the OERR bit. The FIFO is reset when OERR is cleared, which causes data in the buffer to be lost.

The parity error bit PERR (UxSTA<3>) is set if a parity error has been detected in the received data (the last word in the receive FIFO buffer), i.e. the total number of ones in the data is incorrect (odd for EVEN parity mode or even for ODD parity mode). The PERR bit is irrelevant in the 9-bit mode. For the 8-bit mode, prior to reading the data from the receive

FIFO, the FERR and PERR flags should be checked to ensure that the received data are correct.

10.4.1 Receive interrupt

The URXISEL<1:0> (UxSTA<7:6>) control bit determines when the UART receiver generates an interrupt. The UART receive interrupt flag (UxRXIF) is located in the corresponding interrupt flag status, IFS register.

- If URXISEL<1:0> = 00 or 01, an interrupt is generated each time a data word is tranferred from the receive shift register to the receive FIFO buffer. There may be one or more characters in the receive FIFO buffer.
- If URXISEL<1:0> = 10, an interrupt is generated when a word is transferred from the receive shift register to the receive FIFO buffer and as a result, the receive buffer contains 3 or 4 characters.
- If URXISEL<1:0> = 11, an interrupt is generated when a word is transferred from the receive shift register to the receive FIFO buffer and as a result, the receive buffer contains 4 characters, i.e. becomes full.

Switching between the three interrupt modes during operation of the UART module is possible.

The URXDA bit (UxSTA<0>) indicates whether the receive FIFO buffer is empty. This bit is set as long as there is at least one character to be read from the receive buffer. URXDA is a read only bit.

The URXDA and UxRXIF flag bits indicate the status of the UxRXREG register. The RIDLE bit (UxSTA<4>) shows the state of the shift register UxRSR. The RIDLE status bit is a read only bit, which is set when the receiver is IDLE (i.e. the UxRSR register is empty and there is no current data reception). No interrupt logic is tied to this bit, so the user has to poll this bit in order to determine if the UxRSR is IDLE.

10.4.2 Setup for UART reception

When setting up a reception, the following steps should be undertaken:

1. Initialize the UxBRG register for the appropriate baud rate.
2. Set the number of data bits, number of STOP bits, and parity selection by writing to the PDSEL<1:0> (UxMODE<2:1>) and STSEL (UxMODE<0>) bits.
3. If receive interrupts are desired, set the UxTXIE control bit in the corresponding interrupt enable control register (IEC). Specify the interrupt priority using the UxRXIP<2:0> control bits in the corresponding interrupt priority control register (IPC). Select the transmit interrupt mode by writing the URXISEL (UxMODE<15>) bit.
4. Enable the UART module by setting the UARTEN (UxMODE<15>) bit.
5. Receive interrupts will depend on the URXISEL<1:0> control bit settings. If receive interrupts are not enabled, the user can poll the URXDA bit to check the contents of the receive FIFO buffer. The UxRXIF bit should be cleared during initialization.
6. Finally, read data from the receive FIFO buffer via the UxRXREG register. If 8-bit mode is used, read a byte (8-bits). If 9-bit mode has been selected, read a word (16-bits).

Fig. 10.6 shows an example of serial data reception using the UART transmitter. The waveforms for two 8-bit bytes are shown. Fig. 10-7 shows the reception with a receive overrun of the receive FIFO buffer

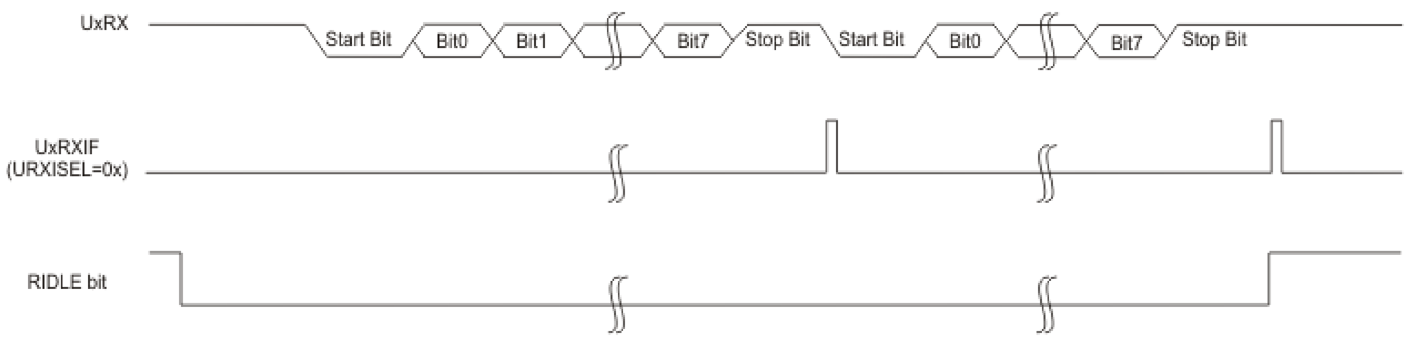


Fig. 10-6 Example of serial data reception of two 8-bit bytes

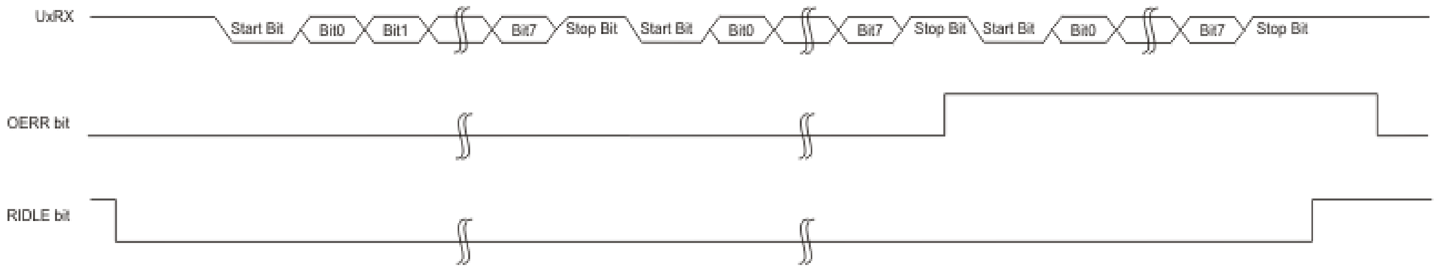


Fig. 10-7 Example of serial data reception with an overrun of the receive FIFO buffer

The UART module is often used for muli-processor communication. In the multi-processor communication typical communication protocols are: data bytes and address/control bytes. A common schematic is to use a 9th data bit to identify whether a data byte is address or data information. If the 9th bit is set, the data is processed as address or control information. If the 9th bit is cleared, the received data word is processed as data associated with the previous address/control byte.

A common multi-processor protocol operates as follows:

1. The master device transmits a data word with the 9th bit set. The data word contains the address of a slave device.
2. The slave devices in the communication chain receive the address word and check the slave address value.
3. The slave device that was addressed will receive and process subsequent data bytes sent by the master device. All other slave devices will discard subsequent data bytes until a new address word (9th bit set) is received.

The UART receiver can be configured to operate in the address detection mode by setting the ADDEN (UxSTA<5>) control bit. In this mode, the receiver will ignore data words with the 9th bit cleared. This reduces the number of interrupts of the UART module, since data words with the 9th bit cleard are not buffered.

To operate in the address detection mode, the UART must be configured for 9-bit data. The ADDEN bit has no effect when the receiver is configured in 8-bit data mode.

The setup procedure for 9-bit transmission is identical to the 8-bit transmit modes, except that PDSEL<1:9> (UxMODE<2:1>) should be set to '11'.

The setup procedure for 9-bit reception is similar to the 8-bit receive modes, except that PDSEL<1:0> (UxMODE<2:1>) should be set to '11'. The receive interrupt mode should be configured by setting the URXISEL<1:0> (UxSTA<7:6>) control bits.

Attention!
If the address detect mode is enabled, the URXISEL<1:0> control bits should be configured so that an interrupt will be generated after every received word, i.e. they should be set to '00' or '11'. Each received data word must be checked in software for an address match immediately after reception.

The procedure for using the address detect mode is as follows:

1. Set the ADDEN (UxSTA<5>) bit to enable address detect. Ensure that the URXISEL control bits are configured to generate an interrupt after rach received word.
2. Check each 8-bit address by reading the UxRXREG register, to determine if the device is being addressed.
3. If this device has not been addressed, then discard the received word.
4. 4. If the device has been addressed, clear the ADDEN bit to allow subsequent data bytes to be read into the receive FIFO buffer and interrupt the CPU. If a long data packet is expected, then the receive interrupt mode could be changed to buffer more than one data byte between interrupts by setting control bits URXISEL<1:0> (UxSTA<7:6>) to '10' or '11'.
5. When the last data byte has been received, set the ADDEN bit so that only address bytes will be received. Also, ensure that the URXISEL control bits are configured to generate interrupt after each received word.

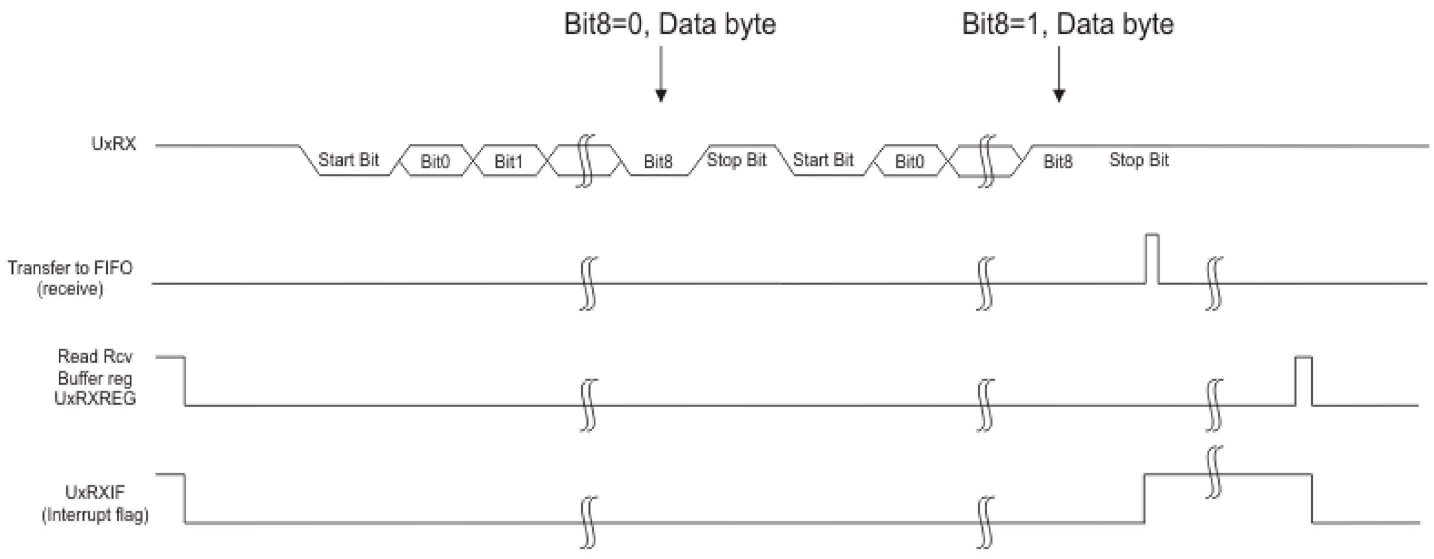


Fig. 10-8 Example of data reception in the address detect mode (ADDEN=1)

The receiver will count and expect a certain number of bit times based on the values programmed in the PDSEL (UxMODE<2:1>) and STSEL (UxMODE<0>) bits. If more than 13 bits at the low logic level occur, the BREAK character has been received. After 13 bits at the low logic level, a STOP bit has to be detected. Then, on the basis of the set FERR flag one can conclude that a BREAK character has been transmitted. If the STOP bit has not been received, the RIDLE status bit is at the low logic level, i.e. the receiver had not detected the end of the message,irrespective of the FERR flag.

Example:
This simple example demonstrates usage of mikroC's UARTx libraries, through a 'loopback' interface. The data being sent to dsPIC through UART and sent back. The example also shows the interconnection of the UART module and the RS-232 transiever and the connection of the UART module to the serial port of a PC.

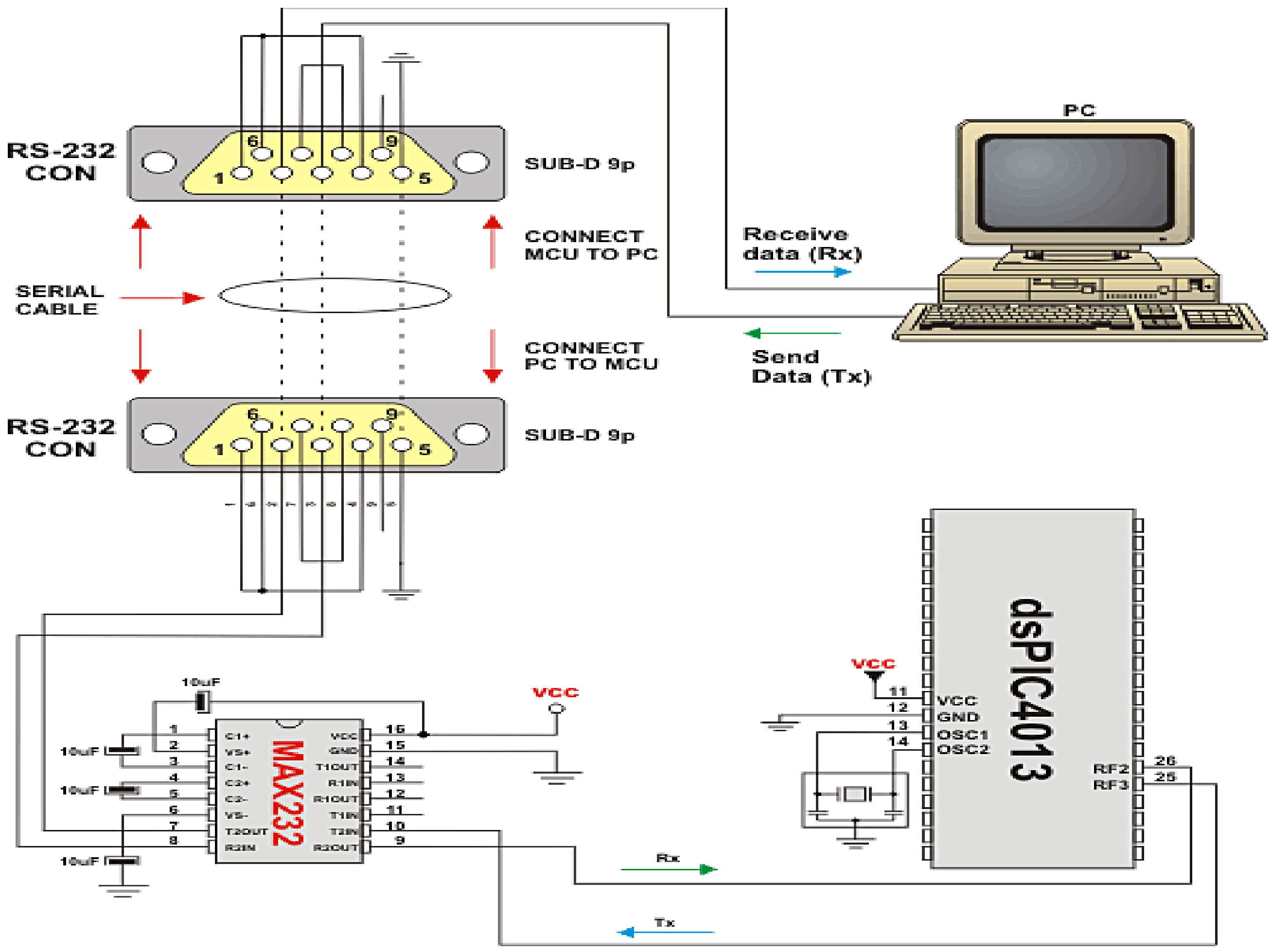


Fig. 10-9 Connection of the UART module to the serial port of a PC via the RS-232 transiever.

```
unsigned rx1;



void main() {



  Uart1_Init(9600);                  // initialize USART module
  //--- un-comment the following lines to have Rx and Tx pins on their
alternate


  //    locations. This is used to free the pins for other module, namely
the SPI.


    U1MODEbits.ALTIO = 1;


    Delay_ms(10);                     // pause for usart lines
stabilization


    rx1 = Uart1_Read_Char();          // perform dummy read to clear the
register




  Uart1_Write_Char('s');            // signal start




  while(1) {


    if (Uart1_Data_Ready())  {     // check if there is data in the buffer


     rx1 = Uart1_Read_Char();      // receive data


     Uart1_Write_Char(rx1);        // send data back


    }


  }


}//~!
```

The program initializes the UART module in the receive and transmit 8-bit format mode, without parity, and with one stop bit. If another format is to be initialized, the function Uart1_Init_Advanced instead of Uart1_Init must be used. In this way it is possible to setup 9-bit data or perhaps 8-bit data with an EVEN or ODD parity bit. Also it is possible to select one or two STOP bits. The function Uart1_Data_Ready the value of the status bit URXDA (U1STA<0>) is read and the presence of data in the receive FIFO buffer is checked. The function Uart1_Write_Char, ASCII character and text are sent respectively via the transmitter of the UART module. Data reception, ASCII character is realized by the function Uart1_Read_Char.

10.5 UART in loopback mode

The UART module has the ability to operate in the loopback mode. This mode allows performing the corresponding self-tests. In order that the UART module operates in the loopback mode, it is required that the LPBACK (UxMODE<6>) control bit is set. In this mode the UxTX output is internally connected to the UxRX input, and the UxRX pin is disconnected from the internal UART receive logic. However, the UxTX pin still functions normally. Sending a message by the UART transmitter causes the reception of this message by the UART receiver. This allows checking if the UART module functions correctly.

To configure the UART module to operate in this mode, the following steps should be taken:

1. Configure UART for the desired mode of operation.
2. Set LPBACK=1 to enable loopback mode.
3. Enable transmission by setting the UARTEN (UxMODE<15>) and UTXEN (UxSTA<10>) control bits.

10.6 Auto baud support

To allow the system to determine baud rates of the received characters, the UxRX input can be internally connected to a selected input capture channel. To operate in this mode, the ABAUD bit (UxMODE<5>) should be set and the UxRX pin is internally connected to the input capture channel. The corresponding ICx pin is disconnected from the input capture channel.

The input capture channel used for auto baud support is device specific. E.g. for dsPIC30F4013 for auto baud support of the UART1 module, the input capture module IC1 is used, whereas for UART2 the input capture module IC2 is used.

This mode is only valid when the UART is enabled (UARTEN=1) and the loopback mode is disabled (LPBACK=0). Also, the user must program the capture module to detect the falling and rising edges of the START bit.

10.7 UART operation in SLEEP abd IDLE modes

When the device enters IDLE mode, the module can continue normal operation if the USIDL (UxMODE<13>) control bit is cleared. If USIDL=1, the module will stop and any transmission or reception in progress will be aborted.

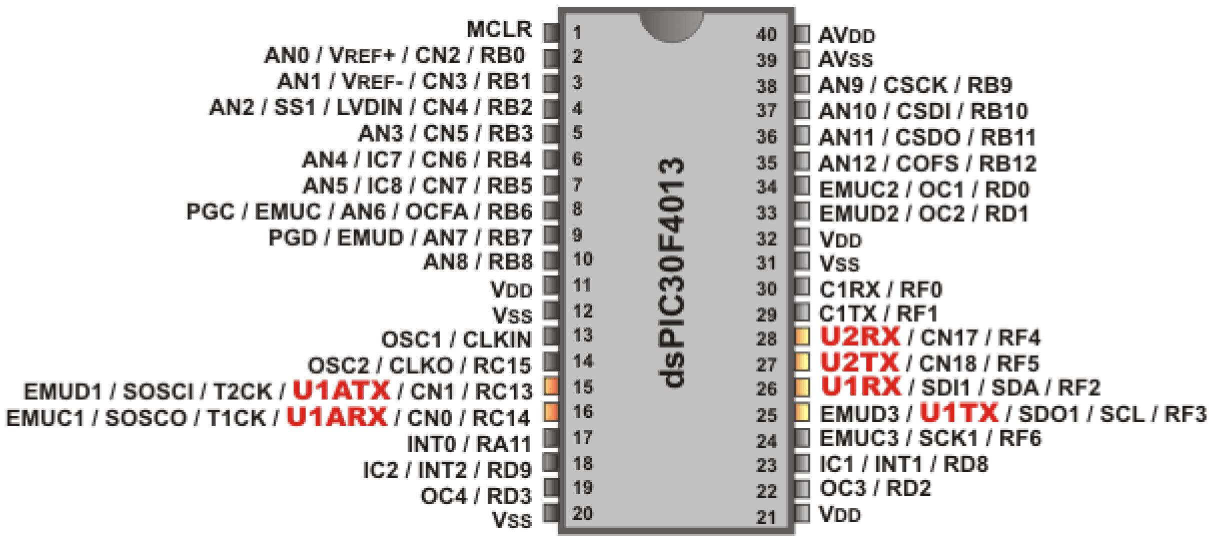


Fig. 10-10a Pinout of dsPIC30F4013

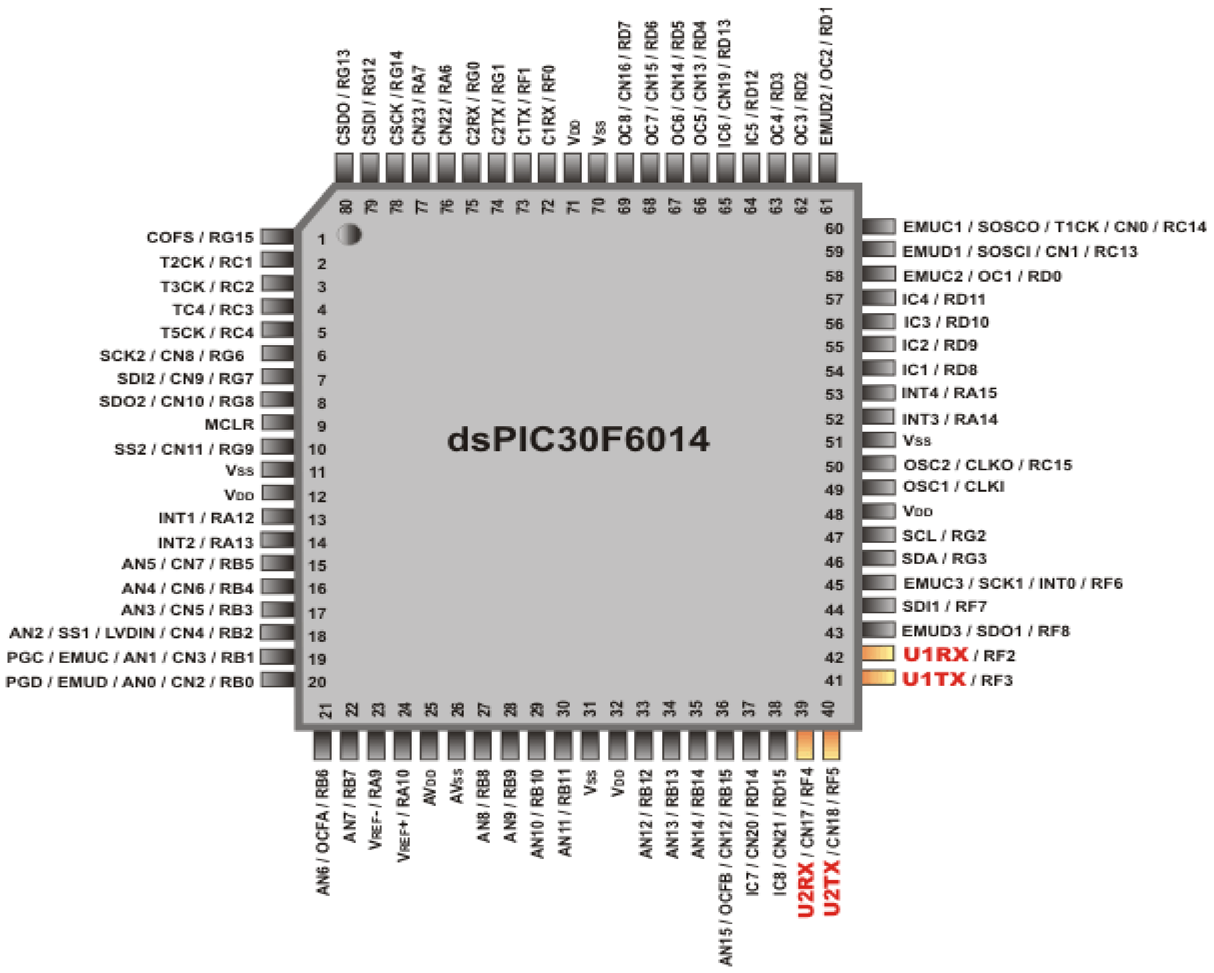


Fig. 10-10b Pinout of dsPIC30F6014A

At the end, a brief description of the registers of the UART module is presented

name	ADR	15	14	13	12	11	10	9
U1MODE	0x020C	UARTEN	-	USIDL	-	-	ALTIO	-
U1STA	0x020E	UTXISEL	-	-	-	UTXBRK	UTXEN	UTXBF
U1TXREG	0x0210	-	-	-	-	-	-	-
U1RXREG	0x0212	-	-	-	-	-	-	-
U1BRG	0x0214	Baud-rate generator prescale						
U2MODE	0x0216	UARTEN	-	USIDL	-	-	ALTIO	-
U2STA	0x0218	UTXISEL	-	-	-	UTXBRK	UTXEN	UTXBF
U2TXREG	0x021A	-	-	-	-	-	-	-
U2RXREG	0x021C	-	-	-	-	-	-	-
U2BRG	0x021E	Baud-rate generator prescale						

Table 10-2 Description of UART module registers

8	7	6	5	4	3	2	1	0	Reset State
-	WAKE	LPBACK	ABAUD	-	-	PDSEL<1:0>		STSEL	0x0000
TRMT	URXISEK<1:0>		ADDEN	RIDLE	PERR	FERR	OERR	URXDA	0X0000
UTX8	Transmit register								0x00uu
Transmit register									0x00uu
-	WAKE	LPBACK	ABAUD	-	-	PDSEL<1:0>		STSEL	0x0000

TRMT	URXISEK<1:0>	ADDEN	RIDLE	PERR	FERR	OERR	URXDA	0X0000
UTX8	Transmit register							0x00uu
Transmit register								0x00uu

Table 10-2 (continued)

UARTEN – UART enable bit (UARTEN=0 UART is disabled, UARTEN=1 UART is enabled)

USIDL – Stop in IDLE mode bit (UISDL=0 continue operation in IDLE mode, USIDL=1 discontinue operation in IDLE mode)

ALTIO – UART alternate I/O selection bit (ALTIO=0 UART communicates using UxTX and UxRX I/O pins, ALTIO=1 UART communicates using UxATX and UxARX I/O pins)

WAKE – Enable wake-up on START bit detect during SLEEP mode bit

LPBACK – UART loop back mode select bit

ABAUD – Auto baud enable bit

PDSEL<1:0> - Parity and data selection bits

- **00** – 8-bit data, no parity
- **01** – 8-bit data, even parity
- **10** – 8-bit data, odd parity
- **11** – 9-bit data, no parity

STSEL – STOP selection bit (STSEL=0 one STOP bit, STSEL=1 two STOP bits)

UTXISEL – Transmission interrupt mode selection bit (UTXISEL=0 interrupt when a character is tranferred to the transmit shift register, UTXISEL=1 interrupt when a character is tranferred to the transmit shift register and the transmit buffer becomes empty)

UTXBRK – Transmit break bit (UTXBRK=0 UxTX pin operates normally, UTXBRK=1 UxTX pin is driven low, regardless of transmitter state)

UTXEN – Transmit enable bit (UTXEN=0 UART transmitter disabled, UTXEN=1 UART transmitter enabled)

UTXBF – Transmit buffer full status bit (UTXBF=0 transmit buffer is not full, UTXBF=1 Transmit buffer is full)

TRMT – Transmit shift register is empty bit (TRMT=0 transmit shift register is not empty, transmission in progress, TRMT=1 transmit shift register is empty, transmission completed)

URXISEL<1:0> - Receive interrupt mode selection bits

- 0x – interrupt flag bit is set when a charatcer is received
- 10 - interrupt flag bit is set when receive buffer is ¾ full (3 locations full)
- 11 - interrupt flag bit is set when receive buffer is full (all 4 locations full)

ADDEN – Address character detect
(ADDEN=0 address detect mode disabled, ADDEN=1 address detect mode enabled)

RIDLE – Receiver IDLE bit
(RIDLE=0 UxRSR not empty, data is being received, RIDLE=1 receiver is IDLE)

PERR – Parity error status bit
FERR – Framing error status bit
OERR – Recive buffer overrun error status bit
URXDA – Receive buffer data available bit
(URXDA=0 receive buffer is is empty, URXDA=1 receive buffer has data, at least one more character can be read)

Chapter11: DSP Engine

Introduction

Digital signal processing (DSP) module is a part of the device specialized for fast execution of the basic mathematical operations (addition, subtraction and multiplication) and for performing automatically accumulation, logical shifting, rounding off and saturation. This module makes the dsPIC30F devices very powerful and considerably extends the scope of their applications.

Processing of digital signals is very demanding. One of the biggest problems is the multiplication required for processing of digital signals. The family of dsPIC30F devices has a hardware implemented multiplier which accelerates considerably the processing. The major part of digital signal processing reduces to calculating the sums of products of two arrays. This module has been designed to allow a fast calculation of the sum of products:

$$A = \sum_{i=0}^{N-1} a_i \bullet b_i$$

Block diagram of the DSP module is shown in Fig.11-1.

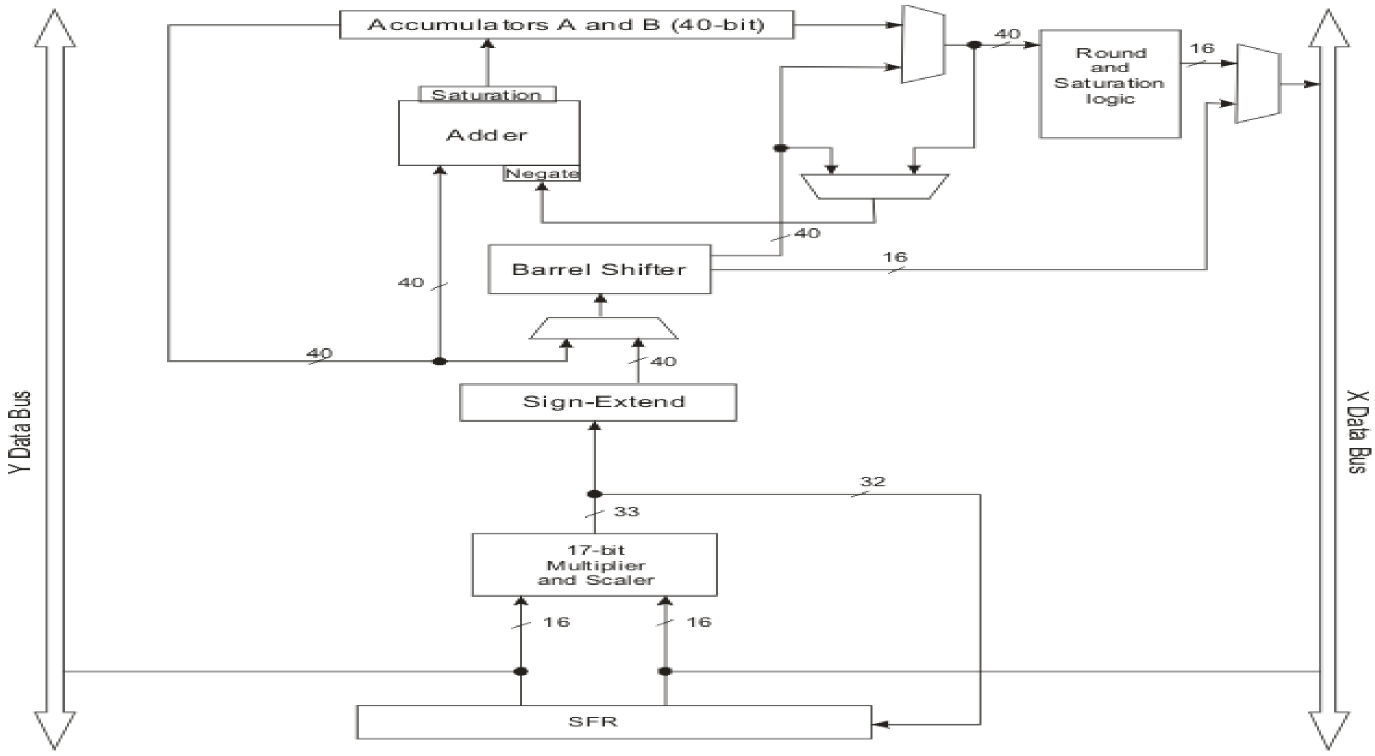


Fig. 11-1 DSP module block diagram

Fig. 11-1 illustrates the realization of the DSP module. In order to calculate the sum of products as fast as possible, the following additions have been made:

1. One data bus for reading the operands (array elements). This gives two data buses, X and Y. The advantage of this approach is that two samples can simultaneously appear at the input of the DSP module, be multiplied and the product added to the partial sum already existing in the accumulator.
2. A very fast hardware multiplier. The multipliers owing to their high complexities and voluminosities are difficult to be built-in in the majority of micoprocessors or microcontrollers. For this reason a multiplier is a part of the micoprocessor or microcontroller only in the applications when it is absolutely necessary, such as digital signal processing. The result of multiplication is a 32-bit quantity.
3. High precision is the property of the DSP module in order to achieve sufficiently good precision while calculating the sum of products of a large number of array elements. Even though 32 bits are sufficient for saving the product of two 16-bit values, 8 bits have been added to increase the precision. This means that the accumulator contains 40-bit data.
4. A barrel shifter serving for automatic hardware shifting of the values from the multiplier, accumulator, or data bus. This eliminates the need for any code for shifting (or multiply/divide by 2n) of any value, thus the processing is accelerated.
5. An adder independent of the multiplier and other parts of the DSP module, which allows a parallel execution of several DSP instructions. E. g. two array members have been multiplied and should be added to the accumulator. While the adder performs adding the value from the multiplier to the value in the accumulator, the multiplier processes next two array elements. This logic allows that processing of one partial sum of products is carried out in one instruction cycle, i.e. pipelining.
6. Hardware rounding off, which is necessary at the end of the calculation of the sum of products, because the intermediate results are kept with a precision which is higher than required in order to reduce the error of the calculation. The rounding off can be convergent or conventional (non-convergent). This part of the DSP module further reduces the length of the corresponding code.
7. Hardware saturation logic which may, but need not, be enabled. This logic prevents the unwanted events at overflow. Namely, 40 bits may sometimes be insufficient, particularly if the summing is performed for two long arrays consisting of the elements having high numerical values. Enabling this logic is recommended since it mitigates the effects of the errors and unwanted phenomena while calculating the sum of products of two large arrays.

11.1 X and Y data buses

$$A = \sum_{i=0}^{N-1} a_i \cdot b_i$$

The process of calculation of the sum of products comprises several steps:

1. The first element of the first array and the first element of the second array (a1, b1) are taken first.
2. The two values are then multiplied.
3. The result is put to the accumulator (and added to the value already saved in the accumulator) A=A+ $a_i \cdot b_i$
   The result of the multiplication of two elements is called partial sum.
4. The process is continued with the next two array elements, until the last two elements have been multiplied.

As can be noticed, the process starts by reading two array elements, the 1st of the first array and the 1st of the second array. Several details need to be considered. Firstly, the array elements being multiplied are from different arrays. This means that the values of these elements are not saved in the adjacent locations. Secondly, the elements always have identical indices, i.e. the address changes are alway identical. E.g. if the arrays consist of 16-bit elements, after the calculation of each partial sum, it is required that the address for the next two elements that is increased or decreased by 2, depending whether the array is saved with the increasing or decreasing index. The change for such arrays will always be ±2. This is important because it allows this change to be caried out by hardware!

Reading the array elements to be multiplied can be accelerated by accessing simultaneously both memory locations. For this purpose it is required that the microcontroller has two data buses, two address generators and that the structure of the memory allows simultaneous access to different locations. The devices of the dsPIC30F family meet these requirements. The data buses are called X and Y bus, as shown in Fig. 11-1. There are two address generators (it is required to calculate simultaneously both addresses to be read). A multiple access to the memory is provided.

The X and Y data buses are considered here. To facilitate the realization, some constraints have to be introduced. Data memory where the elements of the arrays are saved has been split in two sections, X and Y data spaces, as shown in Fig. 11-2.

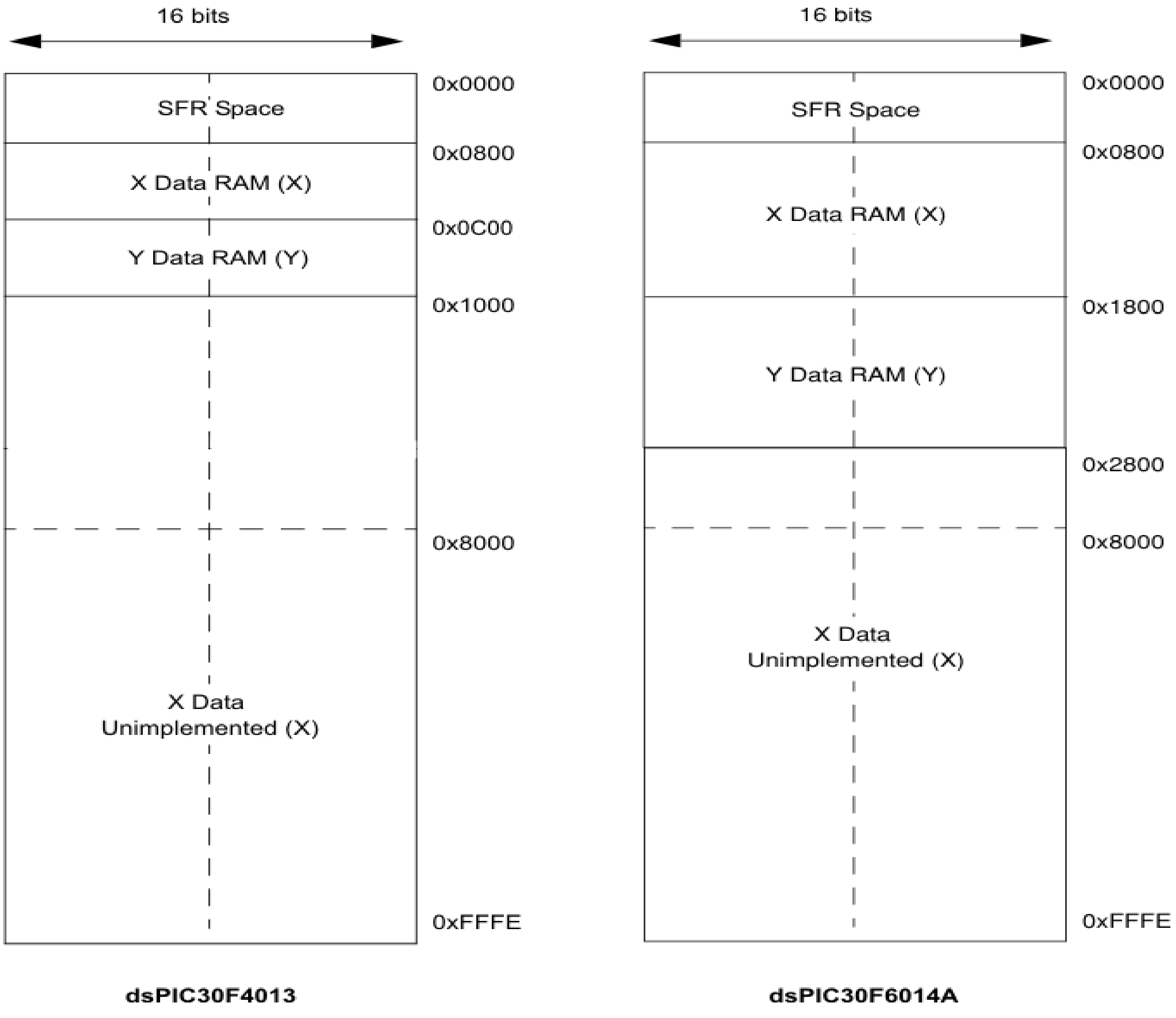


Fig. 11-2 Organizations of data memories of dsPIC30F4013 and dsPIC30F6014A devices

Fig. 11-2 shows the example of data memory organization of a dsPIC30F4013 device. The memory capacity, i.e. the size of the X and Y data spaces are device specific. The space for the special function registers (SFR) remains the same and so does the starting address of the X space.

Splitting data memory to the X and Y data spaces introduces the constraint that each data bus can have access only to one of the spaces. An attempt to access the other space (e.g. an attempt to access X data space via Y data bus) will result in generation of a trap or, if the function for processing a trap has not been specified, in device reset.

The existence of the constraints, when using the DSP instructions has aready been mentioned. The principal contraints when reading data are:

1. The X data bus has access only to X space in the data memory, or extended X space, which will be discussed later,
2. The Y data bus has access only to Y space, which has no extention as the X space,
3. The values to be sent to the multiplier for processing have to be in the general purpose registers W4...W7, specifically W4 and W5 for X space and W6 and W7 for Y space. The general purpose registers for the address pointers indicating the array elements to be read are W8...W11, specifically W8 and W9 for X space and W10 and W11 for Y space.

The practice is as follows:

1. Load the address of the next array element form the X space to the register W8 or W9.
2. Simultaneously, load the address of the next array element form the Y space to the register W10 or W11.
3. Then, simultanmeously read both array elements and load the corresponding values to the W4 or W5 register fot the X array and to W6 or W7 register for the Y array.
4. Values of the registers W8/9 and W10/11 are automatically incremented by 2 (if the array elements are 16-bit quantities).

Of course, all this carried out by the hardware. The code should only provide data concerning address increment (increase/decrease) when calculating the partial sums, the initial and final addresses of the arrays, the registers used for loading the addresses and reading the array elements.

The most frequently used DSP instruction is **MAC**. The following example shows one of the forms of its use.

Example:
MAC W4*W6, A, [W8]+=2, W4, [W10]+=2, W6

The instruction from this example:

1. Multiplies the values in the registers W4 and W6,
2. The result of the mulitplication adds to the accumulator A,
3. From the address in the X space pointed by the register W8 loads the value of the next element to the register W4,
4. After reading the array element in the X space, increments the value of the register W8 to point at the location of the next element of the X array,
5. From the address in the Y space pointed by the register W10 loads the value of the next element to the register W6,
6. After reading the array element in the Y space, increments the value of the register W10 to point at the location of the next element of the Y array.

The hardware specialized for DSP instructions allows that all this is executed in one instruction cycle! As can be seen, the DSP module makes the devices of the dsPIC30F family very powerful. If the device clock is 80MHz, the instruction clock is 20MHz, i.e. in one second 20 milion MAC instructions can be executed each including all six actions listed above!

11.2 PSV management

It is customary that the X space contains stationary arrays, e.g. digital filter coefficients, FFT blocks, etc, whereas the Y space contains dynamic arrays, e.g. samples of the signal being processed or similar. For this reason it was made possible that one section of he program memory is mapped as X space. This means that the coefficients of a digital filter may be saved in the program memory as constants, that this section of the program memory is declared an extention of the X space and that it is accessible as if it was in the data memory. Of course, the addresses of these elements will not be the same as in the program memory, but they will start from the address 0x8000, as can be seen in Fig. 11-2. In this way an additional capacity for saving coefficients has been obtained which is essential particularly for high order filters, FFT algorithms and many other applications. This procedure is known as Program Space Visibility (PSV) Management.

Address generation in PSV management is shown in Fig. 11-3.

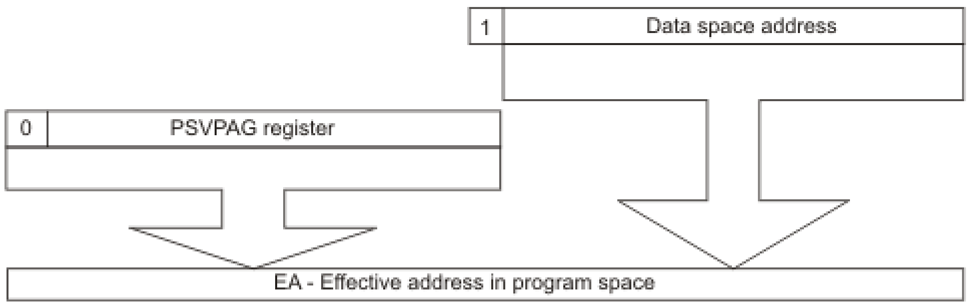


Fig. 11-3 Address generation in PSV management

The element of the array to be read is in the program memory having 24-bit addresses, whereas DSP instructions can operate only with 16-bit addresses. The remaining 8 bits are obtaind from the PSVPAG register. The whole procedure reduces to the PSVPAG register writing the most significant byte od the 24-bit program memory address, PSV management is activated and the array is accessed as if it was in the data memory. The hardware of the DSP module will add 8 bits to each address, as shown in Fig.11-3. In this way a 24-bit address is generated and the array element has been read correctly.

Example:
PSV management is enabled. The array in the program memory starts from the address 0x108000
W1=0x8000
PSVPAG=0x21
In the binary system this is

1000 0000 0000 0000 W1

0010 0001 PSVPAG

$\underline{0}$0010 0001000 0000 0000 0000 = 0001 00001000 0000 0000 0000 = 0x108000 EA

The underlined bit zero is the most significant bit that can be set to zero by hardware (see Fig. 11-3). EA denotes the Effective Address in the program memory.

As Fig. 11-3 shows, of the program memory address only the 15 least significant bits are used and the highest bit is set to logic one to denote that the PSV management is enabled. 8 bits from the register PSVPAG are added to the above 15 bits and the highest bit in the program memory address is logic zero. In this way the addresses of the array elements in the program memory are obtained. All this is done automatically, i.e. when writing a program no attention should be payed to this. All that should be done is to set the corresponding value in the register PSVPAG and activate the PSV management in the register CORCON. The structure of the CORCON register is given at the end of this chapter.

11.3 Multiplier

One of the essential improvements which made possible the execution of an instruction in one instruction cycle is hardware multiplier. The input data are 16-bit quantities and the 32-bit output data are extended to 40 bits in order to facilitate adding to the current value in the accumulator. If this multiplier did not exist, multiplying 16-bit quantities would require 16 instruction cycles which would for many digital signal processing applications be unacceptably long.

The values to be multiplied are via the X and Y data buses fed simultaneously to the input of the multiplier. The output of the multiplier is fed to the to the 40 bits extension block retaining the sign.

By multiplying two 16-bit values one obtains a 32-bit value. However, the aim is not only to multiply but also to calculate the sum of the partial products for the whole array. Therefore, multiplying is only one part and the result is a partial sum. The number of the partial sums will correspond to the length of the array. It follows that the 32 bits will not be sufficient for saving the result because the probability of overflow is high. For this reason it is required to extend the value which will be accumulated. It has been adopted that this extention is 8 bits resulting in the total of 40 bits.

In the worst case that all partial sums are the maximum 32-bit value, one can sum 256 partial sums before overflow. This means that the maximum length of an array consisting of 32-bit elements all of the maximum value is 256. For most applications this is sufficient, but such arrays are very rare and the permissible array lengths are several times longer.

Depending on the values of indiviual bits in the CORCON register, the multiplication may be carried out with signed or unsigned integers or signed or unsigned fractionals formated 1.15 (1 bit for sign and 15 bits for value). The most often used format is fractional (radix).

11.4 Barrel shifter

Barrel shifter serves for automatic shifting the values from the multiplier or accumulator for an arbitrary number of bits to the right or to the left. This operation is often required when scaling a partial or the total sum. The barrel shifter is added in order to simplify the code. Shifting the values is carried out in parallel with the execution of instruction.

The input to the barrel shifter is 40-bit and the output may be 40-bit or 16-bit. If the value from the accumulator is scaled and the result should be fed back to the accumulator, then the output is 40-bit. It is possible to scale the result from the accumulator and save the obtained value in the memory as the final result of the calculation.

11.5 Adder

A part of the DSP module is a 40-bit adder which may save the result in one of the two 40-bit accumulators. The activation of the saturation logic is optional. The adder is required for accumulation of the partial sums. Adding or subtracting of the partial sums is performed automatically as a part of DSP instructions, no additional code is required which allows extermely short time for signal processing.

An example which illustrates the significance of a hardware, independent adder is the previous example of the MAC instruction.

Example:

```
MAC W4*W6, A, [W8]+=2, W4, [W10]+=2, W6
```

The instruction of this example:

1. Multiplies the values from the registers W4 and W6,
2. The result of the multiplication is added to the value in the accumulator,
3. From the address in the X space pointed by the register W8 the value of the next array element is loaded to the register W4,
4. After reading the array element in the X space, the register W8 is incremented to point at the next array element in the X space,
5. From the address in the Y space pointed by the register W8 the value of the next array element is loaded to the register W6,
6. After reading the array element in the Y space, the register W10 is incremented to point at the next array element in the Y space.

It is important that this DSP instruction is executed in one instruction cycle! This means that the whole algorithm for calculating the sum of products consists of loading arrays to the memory, adjusting the parameters of the DSP module (format, positions of the arrays, etc) and then the above instruction is called the corresponding number of times.

Example:

```
CLR A
REPEAT #20
MAC W4*W6, A, [W8]+=2, W4, [W10]+=2, W6
```

The result of the execution of the above program is calling the MAC instruction 21 times (REPEAT means that the next instruction will be called 20+1 times). If the first array elements have been loaded to the registers W4 and W6 and the initial addresses in the data memory or extended data memory (PSV management) loaded to the registers W8 and W10 before the execution of the program, then, upon completion of the REPEAT loop, the accumulator will contain the sum of products of the 20 elements of the two arrays.

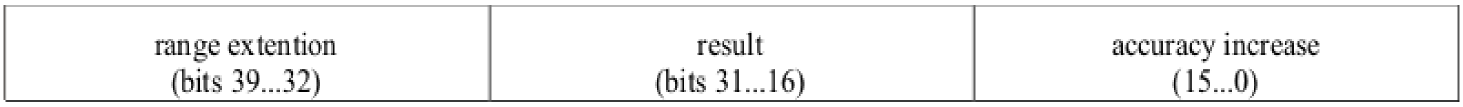
This section of the code occupies 3 locations in the program memory and includes 22 instruction cycles (1 for MOV, 1 for REPEAT and 20 for MAC). If the device clock is 80MHz (20MHz instruction clock), then the program will be executed in 22*50 = 1100ns!

It should be noted that without an independent adder which may carry out the operations simultaneously with the multiplier and other parts of the DSP module, this would not be possible. Then the parallelism in the execution of instructions would not be possible and the execution of one DSP instruction would last at least one clock longer.

Two 40-bit accumulators for saving the partial sums are avialable. These are accumulator A and accumulator B (ACCA and ACCB). The accumulators are mapped in the data memory and occupy 3 memory locations each. The addresses and distribution of the accumulator bits are given at the end of this chapter.

11.6 Round logic

Data accumulation is carried out with 40-bit data. The architecture of the of the dsPIC30F devices, however, is 16-bit, meaning that the conversion to 16-bit values has to be done. A higher number of bits for calculating the sums by DSP instructions is intended for increasing the accuracy and the operating range of values and so reduce the error owing to the finite word length. The purpose of individual bits within the accumulator is presented in Fig. 11-4.



range extention (bits 39...32)	result (bits 31...16)	accuracy increase (15...0)

Fig. 11-4 Purpose of individual bits within accumulator

As can be seen from Fig. 11-4, the upper 8 bits serve for extending the range and the lower 16 bits for increasing the operational accuracy. Increasing the range is sometimes useful as an intermediate step, but the end result should not overrun the basic range. If this occurs, the result will not be correct. The consequences can be mitigated by enabling the saturation logic, but not completely neutralized.

After the calculation is completed, it is required to save the result as a 16-bit quantity. In order to do that, the accuracy of the result has to be reduced. For doing this process automatically, the DSP module is added a block which automatcally rounds off the result during an accumulator write back. From Fig. 11-3 it can be seen that the round logic is placed between the accumulator and X data bus. If the round logic is on (in the CORCON register), by using a separate instruction the result from the accumulator is rounded off and saved in the desired location in the data memory.

The round logic can perform a conventional (biased) or convergent (unbiased) round function.

The conventional round function implies the following. If the MS bit of the accuracy increase bits (bit 15) is logical one, the result will be one step incremented. One step is the least significant positive value that can be saved in the selected format (1 for integer, 1.15 for fractional point), specifically 0.000030517578125. A consequence of this algorithm is that over a succession of random rounding operations, the value will tend to be biased slightly positive.

The convergent round function assuming that bit 16 is effectively random in nature, will remove any rounding bias that may accumulate. If the convergent round function is enabled, the LS bit of the result will be incremented by 1 if the value saved in the accuracy increase 16 bits is greater than 0x08000, or if this value is equal to 0x08000 and bit 16 is one. This algorithm can be readily be explained by using integers. If the middle 16 bits contain an integer, the rounding algorithm will tend towards even integers. This is demonstarted by the examples in Table 11-1.

Value to be rounded	**The result**	**Binary form of the value to be rounded**	**Binary form of the result**
12.75	13	0000 0000 0000 0000 0000 **1100** 1100 0000 0000 0000	0000 0000 0000 **1100**
12.5	12	0000 0000 0000 0000 0000 **1100** 1000 0000 0000 0000	0000 0000 0000 **1100**
13.5	14	0000 0000 0000 0000 0000 **1101** 1000 0000 0000 0000	0000 0000 0000 **1110**

Table 11-1 Covergent round function.
The convergent round function usually gives better results and its use is recommended.

11.7 Saturation logic

When calculating a sum of products of two arrays comprising many elements (more than 256), there is a risk of exceeding the range. In this case, the obtained value is not only inaccurate but also of the opposite sign. These sudden changes of values of a signal (known as glitches) are easily recognized because they violate the characteristics of a signal.

The consequences can be mitigated if the saturation logic is enabled. If, while executing current instruction, an overrun occurs, the hardware saturation logic will load the maximum positive or maximum negative value to the operating accumulator, depending on the previous value loaded to the accumulator. In this way the consequences of a range overrun are mitigated. Fig. 11-6 shows the case of an output sinusoidal signal when an overrun occured, without and with enabled saturation logic.

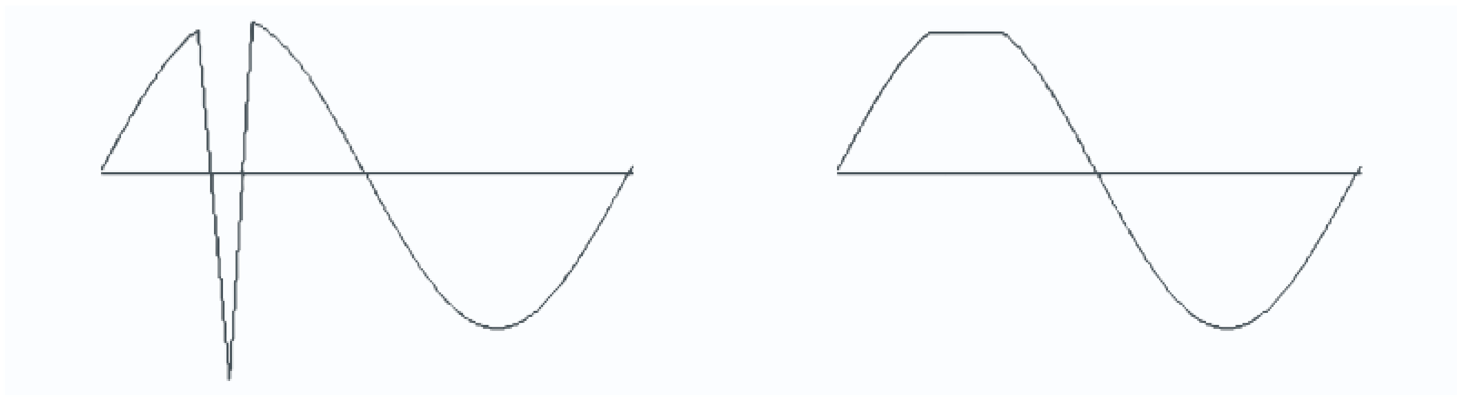


Fig. 11-6 Consequences of range overrun without (left) and with (right) enabled saturation logic

The figure shows that if an overrun occurs and the saturation logic is not enabled, the consequences are greater by far compared to those when the saturation logic is enabled. In the first case a glitch which appears violates considerably the characteristics of the signal. In the second case, owing to the enabled satutration logic, the consequence will only be the unwanted clipping of the crest of the sinusoidal signal, which is much better compared to the first case. With the saturation logic enabled, a lesser overrun corresponds to a lesser consequence, whereas with the saturation logic disabled this does not apply.

There are three modes of operation of the saturation logic: accumulator 39-bit saturation, accumulator 31-bit saturation and write-back saturation logic. In the first case, overrun is allowed until the MS bit (corresponding to the sign in signed operations) is overrun. This is an optimistic version, because it is assumed that by the end of the calculation the signal will decrease to the permitted range. The reason is that the MS 8 bits are the range extention and they are very seldom used so the middle 16 bits contain the final result. This mode is enabled by writing logic one to the ACCSAT bit (CORCON register, bit 4).

A pesimistic version is to enable the saturation logic for the 31 bits when the accumulated value must not overrun the range at any time during the calculation of the sum of products. This mode is enabled by writing logic zero to the ACCSAT bit (CORCON register, bit 4). In case that the saturation logic detects that the current instruction could cause an overrun, the maximum positive value (0x007FFFFFFF) is written to the operating accumulator (A or B) if the accumulator contains a positive value, or the minimum negative value (0xFF80000000) if the accumulator contains a negative value.

If the satuation logic is enabled, at each overrun the bit SA (register SR, bit 13) is set when the saturation logic is enabled for the accumulator A, or SB (register SR, bit 12) when the saturation logic is enabled for the accumulator B. Saturation logic enable for the accumulator A is done by setting the SATA bit (CORCON register, bit 7) to logic one. Similarly, saturation logic enable for the accumulator B is done by setting the SATB bit (CORCON register, bit 6) to logic one.

The third mode of the saturation logic is that the overrun is tested while writing the result from the operating accumulator to a general purpose register (W0...W15). The advantage of this approach is that during calculations it allows using the full range offered by the accumulator (all 40 bits). This logic is enabled only when executing the SAC and SAC.R instructions if the SATDW bit (register CORCON, bit 5) is set to logic one. For the values greater than 0x007FFFFFFF, in the memory (general purpose registers are a part of the data memory) will be written the value 0x7FFFF. Similarly, for the values smaller than 0xFF80000000, in the memory will be written the value 0x8000, representing the smallest negative number that can be expressed by 16 bits.

11.8 DSP instructions

For using the DSP module in an optimum way, it is necessary to konw all DSP instructions. The list of DSP instructions, including the parameter description and application of the instruction is presented in table 11-2.

Instruction	Instruction and parameters	Parameter description	Operation description
MAC	MAC Wm*Wn, Acc	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator	Values of the Wm and Wn registers are multiplied and added to the current value in the operating accumulator (A or B)
MAC	MAC Wm*Wn, Acc, [Wx], Wxd, [Wy], Wyd	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 Wy – W10 or W11 Wyd – W6 or W7	Values of the Wm and Wn registers are multiplied and added to the current value in the operating accumulator (A or B), from the address pointed by the Wx register the value is read and written to the Wxd register, from the address pointed by the Wy register the value is read and written to the Wyd register.
MAC	MAC Wm*Wn, Acc, [Wx]+=kx, Wxd, [Wy]+=ky, Wyd	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 kx – (-6,-4,-2, 2, 4, 6) Wy – W10 or W11	Values of the Wm and Wn registers are multiplied and added to the current value in the operating accumulator (A or B), from the address pointed by the Wx register the value is read and written to the Wxd register, from the address pointed by the Wy register the value is read and written to the Wyd register, the Wx register value is decreased by kx, the Wy register value is decreased by ky.

		Wyd – W6 or W7 ky – (-6,-4,-2, 2, 4, 6)	
MOVSAC	MOVSAC Acc[Wx], Wxd, [Wy], Wyd, AWB	Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 Wy – W10 or W11 Wyd – W6 or W7 AWB – W13 (Acc write-back)	The value from the operating accumulator is saved in the register W13 (AWB - accumulator write back), from the address pointed by the register Wx the value is read and written to the register Wxd, from the address pointed by the register Wy the value is read and written to the register Wyd.
MPY	MPY Wm*Wn, Acc	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator	The values in the Wm and Wn registers are multiplied and written to the operating accumulator.
MPY	MPY Wm*Wn, Acc [Wx], Wxd, [Wy], Wyd	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 Wy – W10 or W11 Wyd – W6 or W7	The values of the Wm and Wn registers are muliplied and written to the accumulator (A or B), from the address pointed by the register Wx the value is read and written to the register Wxd, from the address pointed by the register Wy the value is read and written to the register Wyd.
MPY	MPY Wm*Wn, Acc [Wx]+=kx, Wxd, [Wy]+=ky, Wyd	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 kx – (-6,-4,-2, 2, 4, 6) Wy – W10 or W11 Wyd – W6 or W7 ky – (-6,-4,-2, 2, 4, 6)	The values of the Wm and Wn registers are multiplied and written to the operating accumulator (A or B), from the address pointed by the Wx register the value is read and written to the Wxd register, from the address pointed by the Wy register the value is read and written to the Wyd register, the Wx register value is increased by kx, the Wy register value is increased by ky.

MPY	MPY Wm*Wn, Acc[Wx]-=kx, Wxd, [Wy]-=ky, Wyd	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 kx – (-6,-4,-2, 2, 4, 6) Wy – W10 or W11 Wyd – W6 or W7 ky – (-6,-4,-2, 2, 4, 6)	The values of the Wm and Wn registers are multiplied and written to the operating accumulator (A or B), from the address pointed by the Wx register the value is read and written to the Wxd register, from the address pointed by the Wy register the value is read and written to the Wyd register, the Wx register value is decreased by kx, the Wy register value is decreased by ky.
MSC	MSC Wm*Wn, Acc[Wx], Wxd, [Wy], Wyd	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 Wy – W10 or W11 Wyd – W6 or W7	The values of the Wm and Wn registers are multiplied and subtracted from the curent value in the operating accumulator (A or B), from the address pointed by the Wx register the value is read and written to the Wxd register, from the address pointed by the Wy register the value is read and written to the Wyd register.
MSC	MSC Wm*Wn, Acc[Wx]+=kx, Wxd, [Wy]+=ky, Wyd	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 kx – (-6,-4,-2, 2, 4, 6) Wy – W10 or W11 Wyd – W6 or W7 ky – (-6,-4,-2, 2, 4, 6)	The values of the Wm and Wn registers are multiplied and subtracted from the current value in the operating accumulator (A or B), from the address pointed by the Wx register the value is read and written to the Wxd register, from the address pointed by the Wy register the value is read and written to the Wyd register, the Wx register value is increased by kx, the Wy register value is increased by ky.
MSC	MSC Wm*Wn, Acc[Wx]-=kx, Wxd, [Wy]-=ky, Wyd	Wm – W4 or W5 Wn – W6 or W7 Acc – A or B	The values of the Wm and Wn registers are multiplied and subtracted from the current value in the operating accumulator (A or B), from the address pointed by the Wx register the value is read and written to the

		accumulator Wx – W8 or W9 Wxd – W4 or W5 kx – (-6,-4,-2, 2, 4, 6) Wy – W10 or W11 Wyd – W6 or W7 ky – (-6,-4,-2, 2, 4, 6)	Wxd register, from the address pointed by the Wy register the value is read and written to the Wyd register, the Wx register value is decreased by kx, the Wy register value is decreased by ky.
NEG	NEG Acc	Acc – A or B (operating accumulator)	Acc ← -Acc, the sign of the current value in the accumulator is changed, analogous to the multiplying of the value in the operating accumulator by –1.
REPEAT	REPEAT #lit14	#lit14 – 14-bit unsigned value (0...16383)	The instruction following REPEAT will be executed #lit14+1 times. Even though this is not a DSP instruction, it is very often used when using DSP instructions.
REPEAT	REPEAT Wn	Wn – W0...W15	The instruction following REPEAT will be executed Wn+1 times. Even though this is not a DSP instruction, it is very often used when using DSP instructions.
SAC	SAC Acc, {#Slit4,} Wd	Acc – A or B accumulator {#Slit4,} – optional 4-bit constant Wd – W0...W15	If the optional 4-bit constant is specified, the accumulator value is shifted to the right for the positive value of the constant or to the left if the constant is negative. Then, the obtained value is loaded to Wd.
SAC	SAC Acc, {Slit4,} [Wd]	Acc - A or B accumulator {#Slit4,} – optional 4-bit constant Wd – W0...W15	If the optional 4-bit constant is specified, the accumulator value is shifted to the right for the positive value of the constant or to the left if the constant is negative. Then, the obtained value is loaded to the address in the data memory pointed by the Wd register.
SAC	SAC Acc, {Slit4,} [Wd++]	Acc - A or B accumulator {#Slit4,} – optional 4-bit constant Wd – W0...W15	If the optional 4-bit constant is specified, the accumulator value is shifted to the right for the positive value of the constant or to the left if the constant is negative. Then, the obtained value is loaded to the address in the data memory pointed by the Wd register. After memory write, the value of the register Wd is incremented by 2.
SAC	SAC Acc, {Slit4,} [Wd -]	Acc - A or B accumulator {#Slit4,} – optional 4-bit constant Wd – W0...W15	If the optional 4-bit constant is specified, the accumulator value is shifted to the right for the positive value of the constant or to the left if the constant is negative. Then, the obtained value is loaded to the address in the data memory pointed by the Wd register. After memory write, the value of

			the register Wd is decremented by 2.
SAC	SAC Acc, {Slit4,} [++Wd]	Acc - A or B accumulator {#Slit4,} – optional 4-bit constant Wd – W0...W15	If the optional 4-bit constant is specified, the accumulator value is shifted to the right for the positive value of the constant or to the left if the constant is negative. Then, the value of the register Wd is incremented by 2 and the value obtained by shifting is saved in the address pointed by the Wd register.
SAC	SAC Acc, {Slit4}, [--Wd]	Acc - A or B accumulator {#Slit4,} – optional 4-bit constant Wd – W0...W15	If the optional 4-bit constant is specified, the accumulator value is shifted to the right for the positive value of the constant or to the left if the constant is negative. Then, the value of the register Wd is decremented by 2 and the value obtained by shifting is saved in the address pointed by the Wd register.
SAC.R	The same as for SAC	The same as for SAC	The same as for the SAC instruction except that the value from the accumulator is rounded by the conventional or convergent mode.
SFTAC	SFTAC Acc, #Slit6	#Slit6 – 6-bit constant	Shift the value in the accumulator by #Slit6 bits. If the constant is positive, shifting is to the right, otherwise to the left.
SFTAC	SFTAC Acc, Wd	Wd – W0...W15	Shift the value in the accumulator by Wd bits. If the register Wd is positive, shifting is to the right, otherwise to the left.
CLR	CLR Acc	Acc - A or B accumulator	The value in the operating accumulator is set to zero.
CLR	CLR Acc, [Wx], Wxd, [Wy], Wyd	Acc - A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 Wy – W10 or W11 Wyd – W6 or W7	The value in the operating accumulator is set to zero. From the address in the data memory pointed by Wx the value is read and written to the register Wxd. From the address in the data memory pointed by Wy the value is read and written to the register Wyd.
CLR	CLR Acc, [Wx]+=kx, Wxd, [Wy]+=ky, Wyd	Acc – A or B accumulator Wx – W8 or W9 Wxd – W4 or W5 kx – (-6,-4,-2, 2, 4, 6) Wy – W10 or W11 Wyd – W6 or W7 ky – (-6,-4,-2, 2, 4, 6)	The value in the operating accumulator is set to zero. From the address in the data memory pointed by Wx the value is read and written to the register Wxd. From the address in the data memory pointed by Wy the value is read and written to the register Wyd. The Wx register value is increased by kx, the Wy register value is increased by ky.

Table 11-2 List of DSP instructions with description of operations and parameters.

Table 11-2 shows that some instructions (such as MAC) could have more than one form. All versions of the instructions have not been descibed, but the emphasis was put on the most frequently used versions, in order to illustrate the way of thinking when using DSP instructions.

The structures of individual registers of the DSP module are given in Tables 11-3 to 11-6.

NOTE: Reading of bits which have not been alocated any functions gives '0'.

name	ADR	15	14	13	12	11	10	9	8
CORCON	0x0044	-	-	-	US	EDT	DL<2:0>		

7	6	5	4	3	2	1	0	Reset State
SATA	SATB	SATDW	ACCSAT	IPL3	PSV	RND	IF	0x0020

Table 11-3 Description of the CORCON register

```
US – DSP multiply unsigned/signed control bit
     (1 – unsigned multiplication, 0 – signed multiplication)
EDT – Early DO loop termination control bit. This bit will always read
as’0’.
      1 – Terminate executing DO lop at the end of current loop iteration
      0 – No effect
DL<2:0> - DO loop nesting level status bit
   111 – 7 nested DO loops active
   110 – 6 nested DO loops active
   ...
   001 – 1 nested DO loop active
   000 – 0 DO loops active
SATA – AccA  saturation enable bit
       1 – Accumulator A saturation enabled
       0 – Accumulator A saturation disabled
SATB – AccB  saturation enable bit
       1 – Accumulator B saturation enabled
       0 – Accumulator B saturation disabled
SATDW – Data space write from DSP engine saturation enable bit
       1 – Data space write saturation enabled
       0 – Data space write saturation disabled
ACCSAT – Accumlator saturation mode select bit
         1 – 9.31 saturation (super saturation)
         0 – 1.31 saturation (normal saturation)
IPL3 – CPU interrupt priority level status bit
       1 – CPU interrupt priority level is greater than 7
       0 – CPU interrupt priority level is 7 or less
PSV – Program space visibility in data space enable bit
      (1 – PSV visible in data space, 0 – PSV  not visible in data space)
RND – Rounding mode select bit
      (1- conventional rounding enabled, 0 – convergent rounding enabled)
IF – Integer or fractional multiplier mode select bit
     (1 – integer mode enabled, 0 – fractional mode enabled (1.15 radix))
```

name	ADR	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	Reset State
ACCAU	0x0026	SE								ACCAU								0x0000

Table 11-4a Description of the ACCA register

name	ADR	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	Reset State
ACCAH	0x0024	ACCAH																0x0000

Table 11-4b Description of the ACCA register

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
ACCAL	0x0022	ACCAL																0x0000

SE – Sign extention for AccA accumulator

Table 11-4c Description of the ACCA register

name	ADR	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	Reset State
ACCBU	0x002C	SE								ACCBU								0x0000

Table 11-5a Description of the ACCB register

name	ADR	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	Reset State
ACCBH	0x002A	ACCBH																0x0000

Table 11-5b Description of the ACCB register

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
ACCBL	0x0028	ACCBL																0x0000

SE – Sign extention for AccB accumulator

Table 11-5c Description of the ACCB register

name	ADR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset State
SR	0x0042	OA	OB	SA	SB	OAB	SAB	DA	DC	IPL<2:0>			RA	N	OV	Z	C	0x0000

SE – Sign extention for AccB accumulator

Table 11-6 Description of the SR register

```
OA – Accumulator A overflow status bit
     (1 – accumulator A overflowed, 0 – accumulator A has not overflowed)
OB - Accumulator B overflow status bit
     (1 – accumulator B overflowed, 0 – accumulator B has not overflowed)
SA – Accumulator A saturation ‘sticky’ status bit.
     This bit can be cleared or read but not set to ‘1’.
     1 – accumulator A is saturated or has been saturated at some time
     0 – accumulator A is not saturated
SB – Accumulator B saturation ‘sticky’ status bit.
     This bit can be cleared or read but not set to ‘1’.
     1 – accumulator B is saturated or has been saturated at some time
     0 – accumulator B is not saturated
OAB - OA¦¦OB combined accumulator overflow status bit
      1 – accumulators A or B have overflowed
      0 – neither accumulator A or B have overflowed
SAB - SA¦¦SB combined accumulator ‘sticky’ status bit
      1 – accumulators A or B saturated or have been saturated at some time
      0 – neither accumulator A or B are saturated
DA – DO loop active bit
     (1 – DO loop in progress, 0 – DO loop not in progress)
DC – MCU ALU half carry/borrow bit (1 – a carry-out from the 4th order bit
     (8-bit operations) or 8th order bit (16-bit operations) of the result
occured,
     0 – no carry-out from the 4th order bit (8-bit operations) or
     8th order bit (16-bit operations) of the result occured )
IPL<2:0> - CPU internal priority level status bit.
           These bits are concatenated with the IPL<3> bit (CORCON<3>) to
form
```

```
           the CPU interrupt priority level.
    111 – CPU interrupt priority level is 7(15). User interrupts disabled.
    110 – CPU interrupt priority level is 6(14). User interrupts disabled.
    ...
    001 – CPU interrupt priority level is 1(9). User interrupts disabled.
    000 – CPU interrupt priority level is 0(8). User interrupts disabled.
```

```
RA – REPEAT loop active bit
     (1 – REPEAT loop in progress, 0 – REPEAT loop not in progress)
N – MCU ALU negative bit
    (1- result was negative, 0 – result was non-negative (zero or positive)
OV – MCU ALU overflow bit (1 – overflow occured for signed arithmetic,
     0 – no ovwerflow occured). This bit is used for signed arithmetic (2's
complement).
     It indicates an overflow of the  magnitude which causes the sign bit
to
     change state.
Z – MCU ALU Zero bit (1 – a zero result, 0 – a non-zero result)
C – MCU ALU carry/borrow bit (1 – a carry-out from the MS bit of the result
occured,
    0 – no carry-out from the MS bit of the result occured)
```

11.9 DSP examples

Example 1 – (calculation of the sum of products of two arrays):

$$R = \sum_{i=0}^{19} arr1_i \cdot arr2_i$$

The example shows how the DSP module can be used for fast calculation of the sums of products of two arrays. The elements of the arrays are kept in the data memory. The X and Y spaces have been detailed in Chapter 8.

```
/* dsPIC30F6014A */




char i;


int  arr1[20] absolute 0x0900; // array of 20 signed-integer elements in X
space


int  arr2[20] absolute 0x1900; // array of 20 signed-integer elements in Y
space


                               //   replace 0x1900 with 0x0D00 for
dsPIC30F4013 MCU




void main() {
  TRISD = 0;              // configure PORTD as output


  for (i=0; i<20; i++) {  // init arr1 and arr2

    arr1[i] = i+3;

    arr2[i] = 15-i;

  }


  CORCON = 0x00F1;        // signed computing, saturation for both Acc,
integer computing


  asm {

    mov    #@_arr1, W8    //  W8 = @arr1, point to a first element of array

    mov    #@_arr2, W10   // W10 = @arr2, point to a first element of array


    mov    #0, W4         // clear W4

    mov    #0, W6         // clear W6


    clr    A

    repeat #20

    mac    W4*W6, A, [W8]+=2, W4, [W10]+=2, W6     // AccA = sum(arr1*arr2)


    sftac  A, #-16        // shift the result in high word of AccA
    sac    A, #0, W1      // W1 = sum(arr1*arr2)

    }



  LATD = W1;              // LATD = sum(arr1*arr2)



}
```

Example 1 presents the progam for calculating the sum of products of the array elements. The elements of arr1 are stored in the X space and the elements of arr2 in the Y space of the data memory.

At the start of the program the values of the array elements are initiated.

```
for (i = 0;i < 20;i++){   //init arr1 and arr2

      arr1[i] = i+3;

      arr2[i] = 15-i;

  }
```

Before the calculation starts, it is necessary to set the DSP module for signed-integer computing. This is done by writing 0x00F1 to the register CORCON (CORCON =0x00F1;). At the same time the saturation logic for both accumulators (A nad B) is enabled, even though the accumulator B is will not be used. Table 11-3 gives the meanings of individual bits of the CORCON register.

The next step is writing the initial addresses (addresses of the first array elements) of the **arr1** and **arr2** arrays to the W8 and W10 registers, respectively. It has been decided to use the registers W8 and W10 for specifying the addresses of the next array elements and the registers W4 and W6 for the array elements being multiplied in the current iteration (partial sum).

```
mov #@_arr1, W8

mov #@_arr2, W10
```

Since the addresses of the first array elements are saved in the W8 and W10 registers, the process of multiplying and accumulating the partial sums can be started. Of course, the initial value of the accumulator A is set to zero by **clr A**.

The instruction **MAC**, for calculation of the partial sums and their adding to the current accumulator value should be executed 21 times. The first partial sum will be zero since the values of the registers W4 and W6 are zero. The purpose of the first execution of the **MAC** instruction is to read the values of the first elements from the data memory snd write them to the registers W4 and W6. After that, the instruction **MAC** is executed 20 times, calculating the partial sums which are accumulated in the accumulator A. The instruction **MAC** and the corresponding parameters are described in Table 11-2.

```
repeat #20

mac W4*W6, A, [W8]+=2, W4, [W10]+=2, W6 //AccA:=sum(arr1*arr2)
```

After the instruction MAC has been executed, the result is in the lower 16 bits of the accumulator A. The result could be read directly from AccAL, i.e. from address 0x0022 (see Table 11-4c), but it is regular practice to shift the result to AccAH, i.e. perfom the shift left 16 times and then read the result by using instruction SAC. In this way the consequences of an overflow, if it occurs, will be mitigated. In this case no overflow will occur, nevertheless the result is read in a regular way.

The shift left 16 times is performed by the instruction:

```
SFTAC A, #-16
```

The instruction SFTAC with its parameters is described in Table 11-2. After the result has been shifted 16 places to the left, it is read and saved in the W1 register. This is done by the instruction:

```
SAC A, #0, W1
```

The instruction SAC with its parameters is described in Table 11-2.

NOTE: The instruction SAC reads the results from AccAH (see Fig. 11-4)

Example 2 – (using modulo addressing and PSV management)

The example shows the use of the modulo addressing and PSV management. The result is the sum of array elements of alternated signs:

$$R = \sum_{i=0}^{19} (1)^{i} \bullet \mathrm{arr}_i = arr_0 - arr_1 + arr_2 \ldots$$

The elements are saved in the data memory and the sign (-1, +1) in the program memory.

```
/* dsPIC30F6014A */
```

```
const int Sgn[2] = {1,-1};  // Signes for sum



int arr[14];                // array of 14 signed-integers in Y space (Y
space is default)

int  i;

unsigned int adr2;



void main() {



  TRISD = 0;                // Configure PORTD as output



  for (i=0; i<14; i++)

    arr[i] = i+1;           // init arr



  adr2 = &Sgn;              // dummy line, just for linking Sgn before
usage inside asm block



  MODCON  = 0x8008;         // X modulo addressing, on W8 register

  XMODSRT = adr2;           // XMODSRT points to the start of Sgn array

  XMODEND = adr2+3;         // XMODEND points to the end of Sgn array



  CORCON = 0x00F5;          // Signed computing, saturation for both Acc,
                               // integer computing, PSV managment



  asm {

    mov    #@_sgn, W8       // W8  = @Sgn in X space (mirror),

                               // points to a 1st of 2 elements in Sgn
array

    mov #@_arr, W10         // W10 = @arr, points to a first element of arr

    mov #0, W4              // clear W4

    mov #0, W6              // clear W6

    clr A                   // clear accumulator for computing

    repeat #14              // 15 iterations

    mac W4*W6, A, [W8]+=2, W4, [W10]+=2, W6  // AccA = sum(Sgn*arr)

    sftac A, #-16           // shift the result in high word of AccA

    sac A, #0, W1           // W1 = sum(Sgn*arr)

    }



  LATD = W1;                // LATD = sum(Sgn*arr)



}
```

Example 2 shows the method of using modulo addressing, described in Chapter 8, Section 8.3 and PSV management, described in Chapter 11, Section 11.2.

Constants +1 and -1 for multiplying the elements of the array arr are saved in the program memory. The advantage of this approach is a reduction in using the data memory. It is particularly suitable when several arrays having constant elements should be saved. Then, the use of the program memory is recommended. The data memory should be used when the array elements are not constants (unknown at the moment of compiling) or if the program memory is full (a rare event).

Addresses in the program memory are 24-bit, whereas in the dtata memory are 16-bit. For this reason it is necessary to perform mirroring, by using PSV management, in the upper block of the data memory (addresses above 0x7FFF). The mirroring is performed in two steps:

1. Write in the PSVPAG register the corresponding value (see Figs. 11-3 and 11-7)
2. PSV management is enabled by setting the PSV bit (CORCON<2>, see Table 11-3).

Obtaininmg the value to be written to the PSVPAG register is shown in Fig. 11-7.

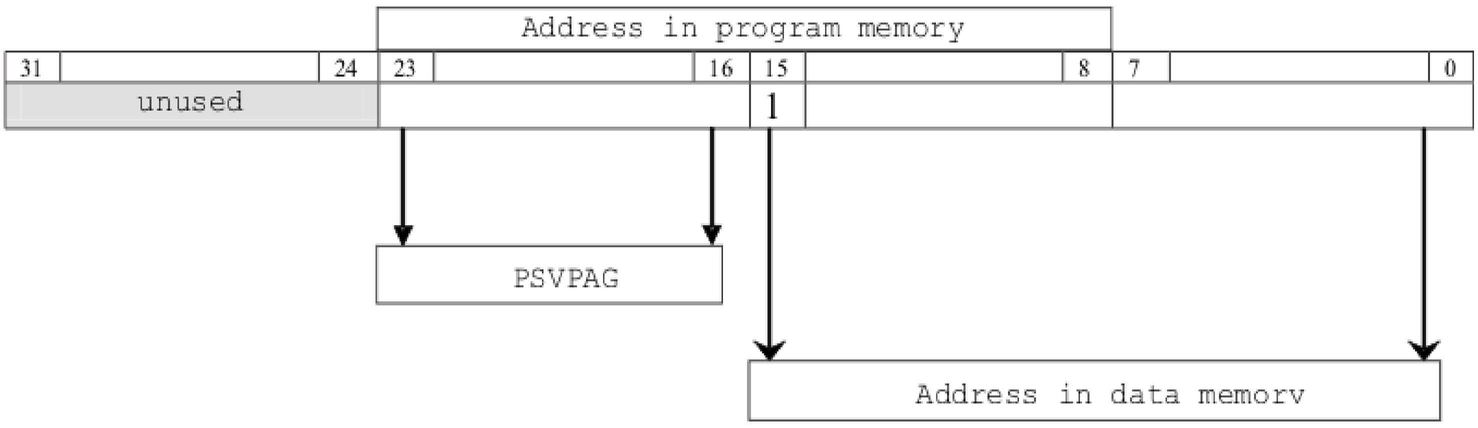


Fig. 11-7 Obtaining the value of the PSVPAG register

Writing to the PSVPAG register and obtaining the effective address (the address of the array mirrored to the data memory) are carried out by the following set of instructions:

```
  ptr =&Sgn;                  //ptr points to Sgn (24-bit address)

  adr = Hi(ptr);              //Get upper Word (16 bits)

  adr2 = adr & 0x00FF;        //Only lower 8 bits are relevant

  PSVPAG = adr2;              //Load PSVPAG

  adr2 = ptr & 0xFFFF;        //get lower Word (16 bits). Mirrored address in
X space

  adr2 = adr2 | 0x0080;       //Upper Data-MEM
```

The variables ptr and adr are 32-bit long (LongInt). When this part of the code is executed, the PSVPAG register will contain the corresponding value and in adr2 will be the address of the first element of the constant array.

The array arr comprises 14 elements and array Sgn only 2. In order to calculate the required sum, modulo addressing should be used for the Sgn array. This is enabled by writing 0x8008 in the MODCON register.

```
MODCON = 0x8008
```

This instruction enables modulo addressing in the X space via the W8 register. The structure of the MODCON register is shown in Chapter 8, Table 8-3.

After modulo addressing has been enabled, it is necessary to define the initial and final addresses of this addressing by writing the corresponding values to the XMODSRT and XMODEND registers. The initial address of the constant array contained by adr2 is written to the XMODSRT register. The address of the last byte of the constant array, i.e. adr2+4-1 (+4 because two elements occupy 2 locations (bytes) each and –1 to obtain the address of the last byte) is written to the XMODEND register.

```
XMODSRT = adr2;

XMODEND = adr2+3;
```

The next step is setting the required bits in the CORCON register. The structure of the CORCON register is shown in Table 11-3. For the signed computing (positive and negative numbers), enabled saturation logic for both accumulators, enabled saturation logic during writing to the data memory, integer computing and enabled PSV management the value 0x00F5 should be written to the CORCON register. Enabling the saturation logic for accumulator B is superfluous, but it is inserted in the example to show that the saturation logic can be enabled for both accumulators simultaneously.

After the corresponding value has been written to the CORCON register, the initialization of the W4, W6, W8 and W10 registers is performed. The registers W4 and W6 are set to zero, whereas in the registers W8 and W10 are written the addresses of the first elements of the arrays Sgn and arr, respectively.

```
CORCON = 0x00F5;

asm

  mov #@_adr2, W8

  mov [W8], W8

  mov #@_arr, W10

  mov #0,W4

  mov #0, W6
```

The initial value of the accumulator is set to zero by the instruction clr A. After that, the computing may start.

```
clr A
```

The repeat loop is used and it is executed 15 times. By performing mac instruction in the W4 and W6 registers the first elements of the arrays are written and then the 14 partial sums are calculated. The consequence of enabling modulo addressing is that the elements of the array Sgn 1,2,1,2,... will be read alternately.

```
repeat #14

mac W4*W6, A, [W8]+=2, W4, [W10]+=2, W6
```

After the loop is completed, the result is in the lowest 16 bits of the accumulator A. This result can be read directly from the address 0x0028. Another approach is used in the example in order to illustrate the use of instructions sftac and sac. These instructions are described in Table 11-2. At first, by using instruction sftac, the result is shifted to the middle 16 bits of the accumulator A. Then, by instruction sac, the result is written to the W1 register and from there forwarded to the port D.

```
sftac A, # - 16

  sac A, #0, W1

end;

LATD = W1;
```

NOTE: Instruction SAC reads the result from AccAH (see Fig. 11-4).

Example 3 – (calculation of mathematical expectation of an array)

In the example it is shown how, by using instruction add, one can select one of the accumulators as a destination and how to use the instruction div for dividing two signed integers.

The instruction divide exists in the compiler and its use is very simple. However, the most efficient use of the DSP module is by using the assembler, so the purpose of this example and of other examples in this chapter is familiarization with the assembler instructions.

For the calculation of mathematical expectation of an array in this example, a function is used. The expression for calculating mathematical expectation is:

$$R = \frac{1}{N}\sum_{i=0}^{N-1} arr_i$$

where N is the number of elements in the array and R mathematical expectation.

```
/* dsPIC30F6014A */



int          arr[15];            // Array of 15 signed-integer elements


unsigned int i, MeanRes;



void MeanVar(unsigned int *ptrArr, unsigned int Len, unsigned int *ptrMean)
{



  CORCON = 0x00F1;               // Signed computing, saturation for both
Acc, integer computing


  asm {


    mov [W14-8], W10             // W10 = ptrArr


    mov [W14-10], W7             // W7 = Len


    sub W7, #1, W2               // W2 = Len-1



    clr A


    repeat W2


    add [W10++], #0, A           // A = sum(arr)



    add W7, #1, A                // A = A + (Len/2) for div's lack of
rounding ...


    sac.r A, #0, W3              // W3 = round(AccA)
    repeat #17                  // 18 iterations of signed-divide. Result
in W0

    div.s W3, W7                // W0 = sum(arr)/Len



    mov [W14-12], W4            // W4 = ptrMean

    mov W0, [W4]                // Mean = Mean(arr)

    }

}



void main() {

  TRISD = 0;                    // Configure PORTD as output



  MeanRes = 0;

  for (i=0; i<15; i++)

    arr[i] = i;                 // Init arr



  MeanVar(&arr, 15, &MeanRes);  // call subroutine



  LATD = MeanRes;               // Send result to LATD

}
```

The main program is very simple. After setting port D as output and initializing the input array, the function for calculating mathematical expectation is called. The result is then sent to port D.

```
TRISD = 0;                    // Configure PORTD as output

MeanRes = 0;

for (i=0; i<15; i++)

  arr[i] = i;                 // Init arr

MeanVar(&arr, 15, &MeanRes);  // call subroutine

LATD = MeanRes;               // Send result to LATD
```

The function for calculating mathematical expectation has only 3 parameters. The first parameter is the address of the first array element (ptrArr = &arr). The second parameter is the number of array elements (Len = 15). The third parameter is the address of the variable where the result, i.e. mathematical expectation, should be written (ptrMean = &MeanRes).

The function consists of three parts:

1. Summation of array elements
2. Division of the result by the number of array elements
3. Saving the result

In order to use the accumulator correctly, the operating conditions of the accumulator should be defined first. This is done by setting the corresponding bits in the CORCON register. Structure of the CORCON reguister is shown in Table 11-3.

```
CORCON = 0x00F1;
```

For the signed computing (positive and negative numbers), enabled saturation logic for both accumulators, enabled saturation logic while writing to the data memory and integer computing the value 0x00F1 should be written to the CORCON register. Enabling the saturation logic for the accumulator B is superfluous, but it is inserted in the example to show that the saturation logic can be enabled for both accumulators simultaneously.

After the CORCON register is set, the address of ther first array element is written to the W10 register. The number of array elements is written to the W7 register.

```
mov [W14-8], W10

mov [W14-10], W7
```

Instruction add should be called as many times as there are elements in the array, i.e. the value in the W7 register should be decremented by 1. Since the number of elements will be required later for performing division, the decrementd value is saved in the W2 register. This is done by the instruction.

```
sub W7, #1, W2
```

Instruction add will be executed the required number of times. The result of each call is the partial sum which is added to the content of the accumulator A. After completion of the loop, the sum of all array elements is in the accumulator.

```
clr A

repeat W2

add [W10++], #0, A
```

To obtain mathematical expectation, the sum of all array elements should be divided by the number of array elements. During division it is not possible to round off the result to the nearest integer. For this reason to the sum of all array elements the value Len/2 is added first. This is the same as adding the value 0.5 to the result, but this is not possible in this case because of integer computing. Adding the value Len/2 is done by the instruction add W7, #1, A. This instruction adds the value of the register W7 shifted one position to the right, which is analogous to divide by two, to the current value in the accumulator A. After that, the value in the accumulator A is read and divided by the number of array elements. This is done by the instruction sac.r A, #0, W3. Instruction sac.r is described in Table 11-2.

```
add W7, #1, A

sac.r A, #0, W3
```

In the family of dsPIC30F devices there is no hardware division. Division is performed by 18 iterations each calling instruction div in the loop. The result will be saved in the W0 register and the remainder in the W1 register. The sum of all array elements is saved in the W3 register and the number of array elements in the W7 register. Therefore, the instruction div.s W3, W7 is called in the loop. After the loop is completed, the result is saved in the W0 register.

```
repeat #17

div.s W3, W7
```

Since the value of mathematical expectation is in the W0 register, it is necssary to write this value to the destination address (third parameter of the function). In this way the obtained value of mathematical expectation is forwarded to the main program for further processing.

```
mov [W14-12], W4


mov W0, [W4]
```

Chapter12: I²C (Inter Integrated Circuit) module

Introduction

The Inter Integrated Circuit (I2C) module is a serial interface primarily intended for communication with other peripheral or microcontroller devices. Examples of the peripheral devices are: serial EEPROM memories, shift registers, display drivers, serial AD converters, etc. In the dsPIC30F family of devices the I2C module can operate in the following systems:

- the dsPIC30F device is a slave device,
- the dsPIC30F device is a master device in a single-master environment (slave may also be active),
- the dsPIC30F device acts as a master/slave device in a multi-master system (bus collision detection and arbitration available).

The I2C module contains independent I2C master logic and I2C slave logic each generating interrupts based on their events. In multi-master systems, the software is simply partitioned into master controller and slave controller. When the master logic is active, the slave logic remains active also, detecting the state of the bus and potentially receiving messages from itself or from other I2C devices. No messages are lost during multi-master bus arbitration. In a multi-master system, bus collision conflicts with other masters in the system are detected and the module provides a method to terminate and then restart the messages.

The I2C module contains a baud-rate generator. The baud-rate generator does not consume other timer resources in the device. The speed of operation of the module may be standard or fast and it may detect 7-bit and 10-bit addresses.

12.1 Operating modes of I2C module

The I2C module may be configured to operate in the following modes:

- I2C slave, 7-bit addressing,
- I2C slave, 10-bit addressing, and
- I2C master, 7- or 10-bit addressing.

A standard I2C interface makes use of two pins, SCL (clock line) and SDA (data line).

The configuration of the I2C module is performed by using the following registers: I2CRV (8 bits), I2CTRN (8 bits), I2CBRG (9 bits), I2CON (16 bits), and I2CADD (10 bits). Functional block diagram of the I2C module is shown in Fig. 12-1.

The I2CCON and I2CSTAT are 16-bit control and status registers, respectively. The content of the I2CCON control register may be read or written and so is the content of the 10 higher bits of the I2CSTAT status register. The lower 6 bits of the status register are read only.

The I2CRSR shift register performs parallel-series conversion of data. Data to be transmitted are written in the I2CTRN buffer register, whereas received data are read from the I2CRCV buffer register. Address of a slave device on the I2C bus is written in the I2CADD register. The control bit A10M (I2CCON<10>) denotes whether the 7-bit or 10-bit addressing is used. Baud-rate is defined by the I2CBRG register, i.e. it holds the baud-rate generator reload value for the I2C module baud-rate generator.

NOTE: In receive operation the shift register I2CRSR and the buffer register I2CRCV create a double-buffered receiver, i.e. when a complete message is received by the I2CRSR register, the byte transfers to the I2CRCV register and the interrupt request for the I2C module is generated. While transmitting a message the transmit buffer is not double-buffered.

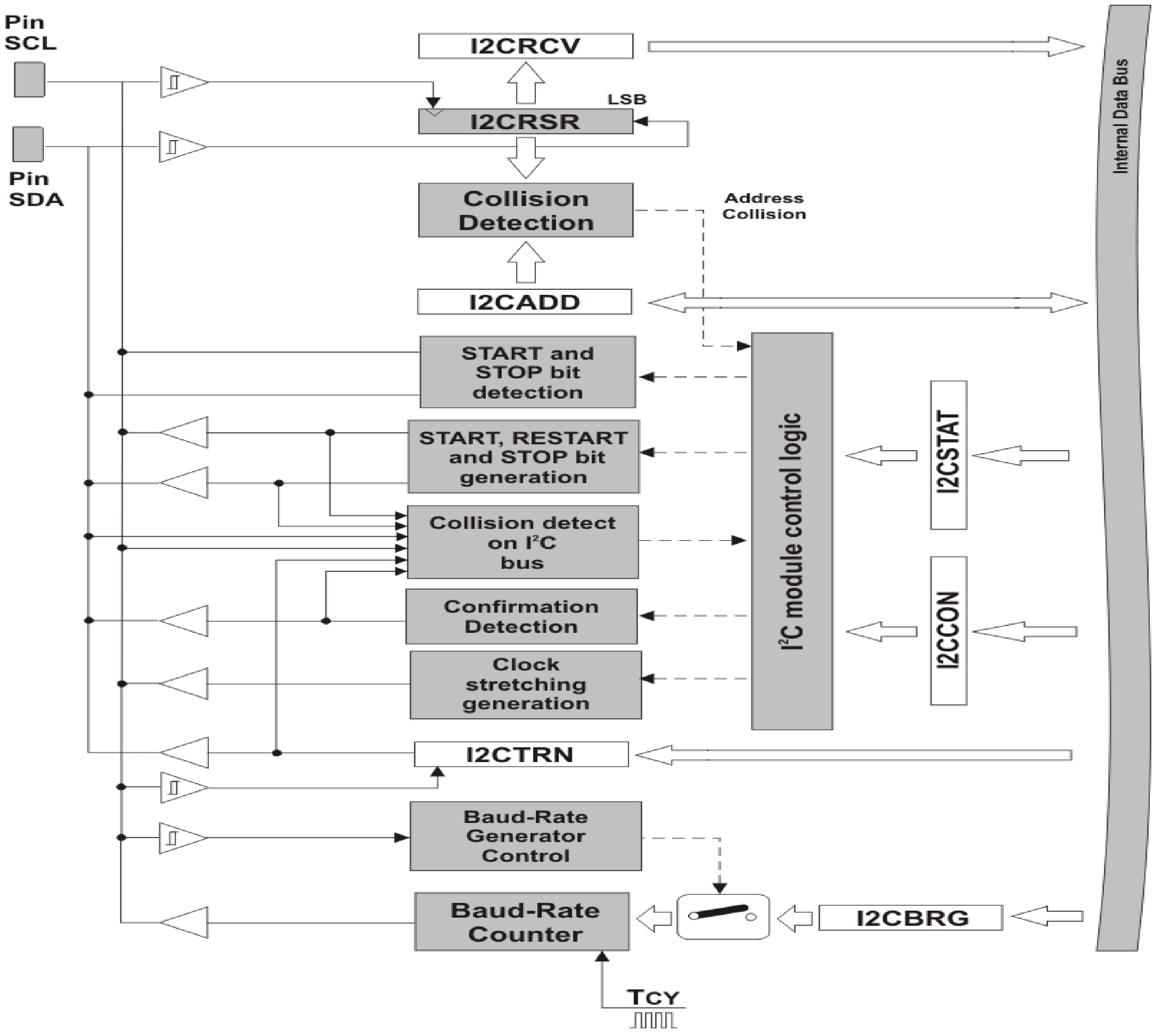


Fig. 12-1 Functional block diagram of I2C module

12.1.1 I2C slave mode with 7-bit addressing

The 10-bit I2CADD register holds the slave device address. If the control bit A10M (I2CCON<10>) is cleared, address in the I2CADD is interpreted as a 7-bit address. When an address from a master device is received, only 7 lower bits are compared against the register I2CADD. If the control bit ADD10 (I2CCON<10>) is set, the value of the received address is compared in two cycles. At first, the 8 higher bits of the address are compared with the value '11110 A9 A8' (where A9 and A8 are two most significant bits of the device address held in the I2CADD register). If coincidence occurs, the lower 8 bits of the register I2CADD are compared with the 8 lower bits of the received address.

The segments of an address usable in a standard I2C communication are specified by the standard I2C protocol. Table 12-1 shows the ranges of I2C addresses supported by the family of dsPIC30F devices together with the corresponding applications.

Address	**Description**
0x00	General call address or starting byte

0x01 – 0x03	Range of reserved addresses
0x04 – 0x77	Range of valid 7-bit addresses
0x78 – 0x7B	Range of valid 10-bit addresses
0x7C – 0x7F	Range of reserved addresses

Table 12-1 Standard I2C addresses supported by the family of dsPIC30F devices

When in the slave mode, after setting the I2CEN (I2CCON<15>) enable bit, the slave function will begin to monitor the I2C bus to detect a START event. Upon deteting a START event, 8 bits are loaded to the I2CRSR shift register and the receved address is compared with the one from the I2CADD register. In the 7-bit address slave mode (A10M = 0) the bits I2CADD<6:0> are compared with the I2CRSR<7:1> bits. The bit I2CRSR<0> is ignored. The bits loaded to the shift register are sampled by the rising edge of SCL the clock.

If an address match occurs, the slave sends an ACKNOWLEDGE of the 9-bit message, and an interrupt of the I2C module is generated (the flag SI2CIF is set) on the falling edge od the ninth bit (ACK – acknowledegement). The address detection is of no influence neither on the content of the receive buffer register I2CRCV nor on the value of the bit RBF (I2CSTAT<1>) which indicates that there is a new value in the receive buffer register.

A typical standard I2C message consists of the START bit, address bits, R/W bit, acknowledge bit, data bits (eg. in an I2C communication with the EEPROM memory within data bits are the location address bits and the contents bits which are read out or written in), and STOP bit.

A slave device transmit process is carried out as follows. Upon reception of the address part of the message containing the R/W bit, the device enters the transmit mode. It sends the ACKNOWLEDGE bit (ACK) and keeps the SCL clock signal low until the message byte is written to the transmit buffer I2CTRN. Then, the SCL clock signal is released, the bit SCLREL is set (I2CCON<12>), and the 8-bit message is sent from the I2CRSR shift register. The bits are sent on the falling edge of the SCL clock so that the SDA pin data are valid when the SCL closk is high. The interrupt request is generated on the falling edge of the ninth clock period irrespective of the status of the ACKLOWLEDGE (ACK) bit of the master device.

The receive sequence of a slave device comprises the following steps. Upon receiving the address part of the message which contains the R/W bit cleared, the slave device is initiated for the receive mode. The incoming bits at the pin SDA are sampled on the rising edge of the SCL clock signal. After receiving 8 bits, if the receive buffer register I2CRCV is not full or the flag RBF (I2CSTAT<1>) is not cleared or the overflow status bit (I2COV (I2CSTAT<6>) of the receive buffer register I2CRCV is not set, the received bits contained by the shift register are transferred to the receive buffer register I2CRCV. Upon completion of the data transfer, the ACKNOWLEDGE bit ACK is generated as the ninth bit. If the flag RBF (I2CSTAT<1>) is set, the ACKNOWLEDGE bit ACK is not sent, but the interrupt request for the I2C is generated. In the case of the receive buffer overflow, the content of the shift register I2CRSR is not transferred to the receive buffer I2CRCV.

Attention!
The status overflow bit of the receive buffer I2COV (I2CSTAT<6>) is hardware set, but the user can clear it by the software. If the bit I2COV (I2CSTAT<6>) is not cleared, the I2C module will operate irregularly, i.e. a recevied message is written to the receive buffer register I2CRCV if RBF was cleared, but the bit acknowledge ACK is not sent to the master device.

12.1.2 I2C slave mode with 10-bit addressing

The basic process of receiving and transmitting messages in the slave mode with 10-bit addressing is the same as in the slave mode with 7-bit addressing, the only difference is that the method of comparing addresses is much more complicated. The I2C standard specifies that after the START bit, any slave devise has to be addressed by two bytes. The control bit A10M (I2CCON<10>) is set meaning that a 10-bit address is in the I2CADD regsiter. After the START bit, the received data in the shift register I2CRSR<7:1> are compared with the expression '11110 A9 A8' containing two most significant bits of the device address, A9 and A8. If the result of the comparison is positive and the received R/W bit is cleared, an interrupt request is generated. At the same time the bit ADD10 (I2CSTAT<8>), which denotes partial match of the sent address and the slave address, is cleared. If the sent address and the slave address do not match or if the received R/W bit is set, the status bit ADD10 is cleared and the I2C is returned to the IDLE state. If there is partial match with the upper part of the address of the slave device, the lower part of the address is received and compared with the lower part of the address register I2CADD (I2CADD<7:0>). If a full match occurs, an interrupt request for the I2C module is generated and the status bit ADD10 (I2CSTAT<8>), indicating a full match of the 10-bit addresses, is set. If a full match does not occur, the status bit ADD (I2CSTAT<8>) is cleared and the module returns to the IDLE state.

In the case when a full match of all 10 bits is detected, the master device may send the part of the message defining whether the slave is to receive data or trasmit data to the master. If the master repeats the START bit, the upper address bit is cleared and the R/W bit is set in order to avoid that the STOP bit is generated, and the master performs initialization of the slave device for the operation transmit data. The synchronization of the read and transmit data to the master is performed in the slave device by stretching the SCL clock signal.

Whenever the slave device is to transmit data to the master, in both 7-bit or 10-bit addressing, stretching of the clock signal SCL is implemented by inserting the SCLREL (I2CCON<12>) bit on the falling edge of the ninth period of the clock. In the case when the transmit buffer is empty, the status bit TBF (I2CSTAT<0>) is cleared. Stretching of the SCL clock signal in the slave mode at data transmition is always performed irrespective of the state of the STREN (I2CON<6>) control bit which enables stretching of the SCL clock. This is not the case in the slave mode at data reception. The clock synchronization occurs after the ninth period of the SCL clock. Then, the slave device samples the value of the ACK acknowledgement bit (active low) and checks the value of the TBF status bit. If the ACK is low (the acknlowledge ACK bit active) and the TBF (I2CSTAT<0>) status bit is cleared, the SCLREL (I2CCON<12>) bit is cleared automatically. Setting the SCLREL (I2CCON<12>) bit low causes setting the external SCL line low. By the interrupt routine the user has to set the bit SCLREL (I2CCON<12>) before the slave device starts transmitting the message to the master. In this way the user is given the chance that in the interrupt routine writes data to be transferred to the I2CTRN transmit buffer register before the master device initiates a new transmit sequence.

Attention!
If the user writes data to be transferred to the I2CTRN transmit buffer register and sets the TBF bit before the falling edge of the ninth period of the SCL clock, the SCLREL (I2CCON<12>) bit will not be cleared and stretching of the SCL clock signal will not occur. Also, it shoud be noted that the SCLREL (I2CCON<12>) bit can be set by the software irrespective of the TBF bit.

In essence, the STREN (I2CON<6>) control bit primarily serves for enabling stertching of the SCL clock signal during reception in the slave mode. When the STREN (I2CON<6>) bit is set, the pin of the SCL clock signal is kept low at the end of each receive sequence in the slave mode. Stretching of the clock signal is performed after the ninth period of the SCL clock signal during a receive sequence. On the falling edge of the ninth period of the SCL clock signal the value of the ACK ACKNOWLEDGE bit (active low) is sampled and the value of the TBF status bit is checked. If the ACK is low (bit ACK active) and the RBF (I2CSTAT<1>) status bit set, the SCLREL (I2CCON<12>) bit is cleared automatically. Setting the SCLREL (I2CCON<12>) bit low causes setting the external SCL line low. By the interrupt routine the user has to set the bit SCLREL (I2CCON<12>) before the slave device starts receiving data from the master. In this way the user is given the chance that in the interrupt routine reads out received data from the I2CRCV receive buffer before the master device initiates a new receive sequence. This prevents the overflow of the I2CRCV receive buffer register.

Attention!
If the user reads out received data from the I2CRCV receive buffer register and clears the RBF bit before the falling edge of the ninth period of the SCL clock signal, the SCLREL (I2CCON<12>) bit will not be cleared and no stretching of the SCL clock signal will occur. Also, the SCLREL (I2CCON<12>) bit can be set by the software irrespective of the RBF bit. Clearing the RBF bit by the user has to be done carefully during the interrupt routine to avoid overflow of the I2CRCV receive buffer register.

12.1.3 I2C master mode

A master device differs from a slave device in that it generates the system SCL serial clock, and START and STOP bits. Each data transfer is ended by a STOP bit or by the repeated START bit. A repeated START bit also means the beginning of a new data transfer, but the I2C bus is still busy by the current communication session.

In the master mode serial data are transferred via the SDA pin in synchronism with the SCL clock signal. The firstbyte to be sent contains the 7-bit address of the slave (the device for whom the message is intended) and one bit for data direction R/W. In the case of data transmit the R/W bit is cleared (logic zero). The STARTand STOP bits are sent in order to recognize the beginning and end of a communication session, i.e. of a data transfer.

In the master mode when receiving data, the 7-bit address of the slave (the device for whom the message is intended)and one bit for data direction R/W are sent first. In the case of data receive the R/W bit is set (logic one). Serial data are transmitted or received via pin SDA in synchronism with the SCL clock signal. Serial data are received 8 bits per cycle and each cycle is ended by sending the ACKNOWLEDGE ACK bit. The START and STOP bits are sent in order to recognize the beginning and end of a communication session.

In the master mode the transmission of a 7-bit address or of the second part of a 10-bit address is performed by writing the desired value to the transmit buffer register I2CTRN. The user writes the desired value to the transmit buffer register only when the I2C module is in the WAIT state. By writing data to the transmit buffer register I2CTRN, the status flag TBF (I2CSTAT<0>) is set and the baud-rate generator is enabled to start counting and start data transmission. Each data bit or address bit from the shift register is sent to the SDA pin on the falling edge of the SCL clock signal. When the device is transmitting data, the transmit status flag TRSTAT (I2CSTAT<14>) is set.

The reception in the master mode is enabled by setting, using the software, the RCEN (I2CCON<11>) control bit. In order to set the RCEN (I2CCON<11>) control bit, the I2C module has to be in the IDLE state. If this is not observed, the receive enable control bit RCEN (I2CCON<11>) will be ignored. The baud-rate generator starts counting and when it reaches the preset count it is reset, then the state of the SCL clock signal is changed, the ACKNOWLEGDE ACK bit is set, and the value of the SDA line is sampled on the rising edge of the clock signal and transferred to the shift register I2CRSR.

12.2 Baud-rate generator

In the master mode of the I2C module the value of the baud rate is defined by the preset value of the baud-rate counter BRG in the I2CBRG register. The baud-rate counter counts down from the preset value to zero. When zero is reached, the preset value BRG is written to the counter again. In the case when the clock arbitration occurs, writing the preset value BRG to the baud-rate counter is performed when the pin SCL is high.

According to the I2C standard, the frequency of the clock signal FSCK is 100kHz or 400kHz. The user has the optionofspecifying the baud rate up to 1MHz. The following expression defines in the I2CBRG register the value of the desired baud rate:

$$I2CBRG = \left(\frac{F_{CY}}{F_{SCK}} - \frac{F_{CY}}{1111111}\right) - 1$$

The arbitration of the clock signal is performed each time when the master device releases the SCL pin (the SCL pin is set high) during reception, transmission, or the repeated START state or STOP state. When the master device releases the SCL pin, the baud-rate generator does not operate. The preset value BRG is written each time the SCL pin is set high while the baud-rate generator does not operate. Then, the baud-rate counter starts counting and generates the SCL clock signal.

12.3 I2C module interrupts

The I2C module can generate two interrupts by setting the flag bits MI2CIF (IFS0<14> - I2C master interrupt flag) for a master interrupt request and SI2CIF (IFS0<13> - I2C slave interrupt flag) for a slave interrupt request. An I2C master interrupt request MI2CIF (IFS0<14>) is generated on completion a master message event (transmission or reception). A slave interrupt request SI2CIF (IFS0<13>) is generated when a message intended for a slave device is detected.

12.4 Slope control

In the case of the fast mode (400kHz), the standard I2C communication interface requires the slope control on pins SDA and SCL. The control bit DISSLW allows that, if desired, the slope control is skipped in order to allow baud rates up to 1MHz.

12.5 Intelligent peripheral management interface (IPMI) support

By setting the control bit IPMIEN the use of the Intelligent peripheral management interface (IPMI) support is enabled which allows that the I2C module covers all defined addresses.

12.6 General call address support

All devices on the I2C bus are addressed by using the general call address. When the general call address is used, all device on the I2C bus which receive a message with this address should (at least theoretically) confirm the reception by the ACKNOWLDGE ACK bit. In essence, the general call address is one of the addresses reserved by the I2C protocol. These are usually all '0' with the direction bit R/W=0.

In order to enable the recognition of the general call address, the control bit GCEN (I2CCON<15>) which enables general call address has to be set.

12.7 Operation of I2C module in SLEEP and IDLE states

12.7.1 Operation of I2C module in SLEEP state

When the microcontroller enters SLEEP state, all clock sources are off, consequently the baud-rate generator stops operating and stays low (logic zero). If entering the SLEEP state occurs during transmission of a message, the current transmission is interrupted and the transmitted data are lost. Similarly, if entering the SLEEP state occurs during reception of a message, the current reception is interrupted and the received psrt of the message is lost.

12.7.2 Operation of I2C module in IDLE state

In the IDLE state the I2C module may continue operating if the control bit I2CSIDL (I2CCON<13>) is not set. If the bit I2CSIDL (I2CCON<13>) is set, the I2C module behaves like in the SLEEP state.

Example:

This example shows how to use the specialized I2C library of the mikroC compiler for dsPIC microcontrollers which allows easy initialization of the I2C module, writing, and reading data from the transmit and receive buffers of the I2C module, respectively. Specifically, the example shows the method of connecting the I2C module to the serial I2C EEPROM memory 24C02.

```
unsigned dAddr;



void main() {

  ADPCFG = 0xFFFF;

  PORTB = 0;

  TRISB = 0;



  dAddr = 0x02;
  I2c_Init(100000);

  I2c_Start();                 // issue I2C start signal

  I2c_Write(0xA2);             // send byte via I2C  (command to 24cO2)

  I2c_Write(dAddr);            // send byte (address of EEPROM location)

  I2c_Write(0xF4);             // send data (data to be written)

  I2c_Stop();



  Delay_100ms();



  I2c_Start();                 // issue I2C start signal

  I2c_Write(0xA2);             // send byte via I2C  (device address + W)

  I2c_Write(0x02);             // send byte (data address)

  I2c_Restart();               // issue I2C signal repeated start

  I2c_Write(0xA3);             // send byte (device address + R)

  PORTB = I2c_Read(1);         // Read the data (NO acknowledge)

  I2c_Stop();



}//~!
```

The function I2C_Init initializes the I2C module, i.e. baud-rate. The function I2C_Start sets the conditions for START of the communication session on the I2C bus (defines the beginning of the message). The functions I2C_Write and I2C_Read enable that a microcontroller of the family dsPIC30F as a master can write or read from a peripheral having the I2C interface. In addition to these functions, also significant is the function I2C_Stop setting the state at the end of the I2C communication session (defines the end of the message). The I2C library of the mikroC compiler for dsPIC microcontrollers contains also the functions I2C_Repeated_Start and I2C_Is_Idle generating the state of a repeated start and the state of waiting of the I2C module, respectively. Fig. 12-2 shows the connection of a microcontroller of the family dsPIC30F to a serial I2C EEPROM memory 24C02.

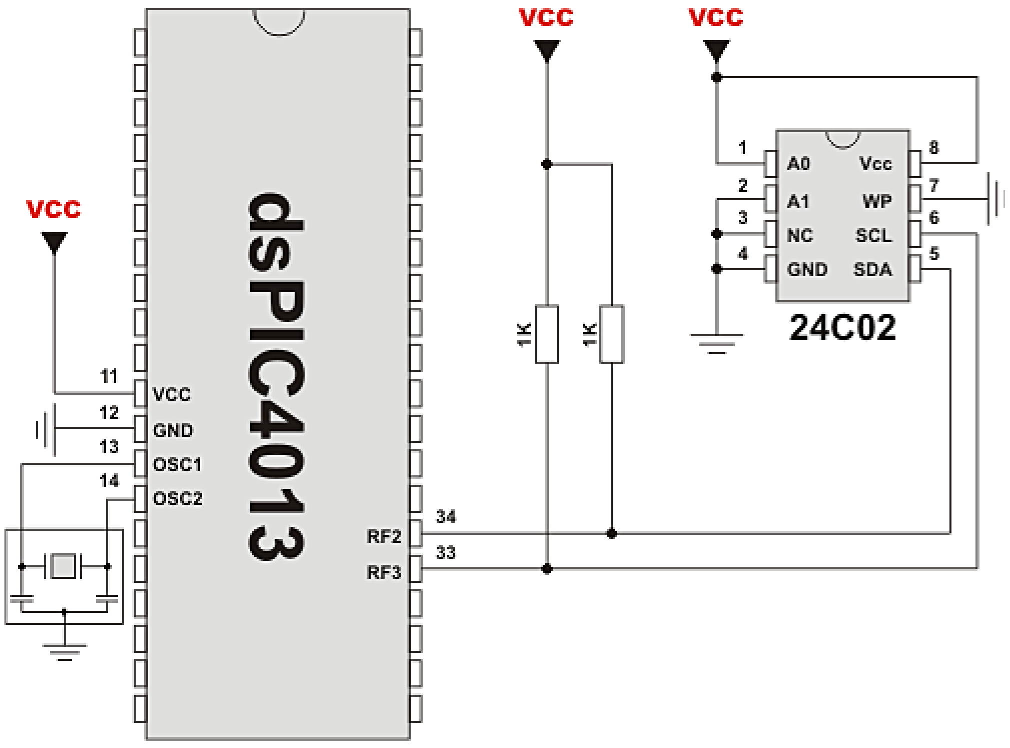


Fig. 12-2 Interconnection between a dsPIC30F device and a serial I2C EEPROM memory 24C02

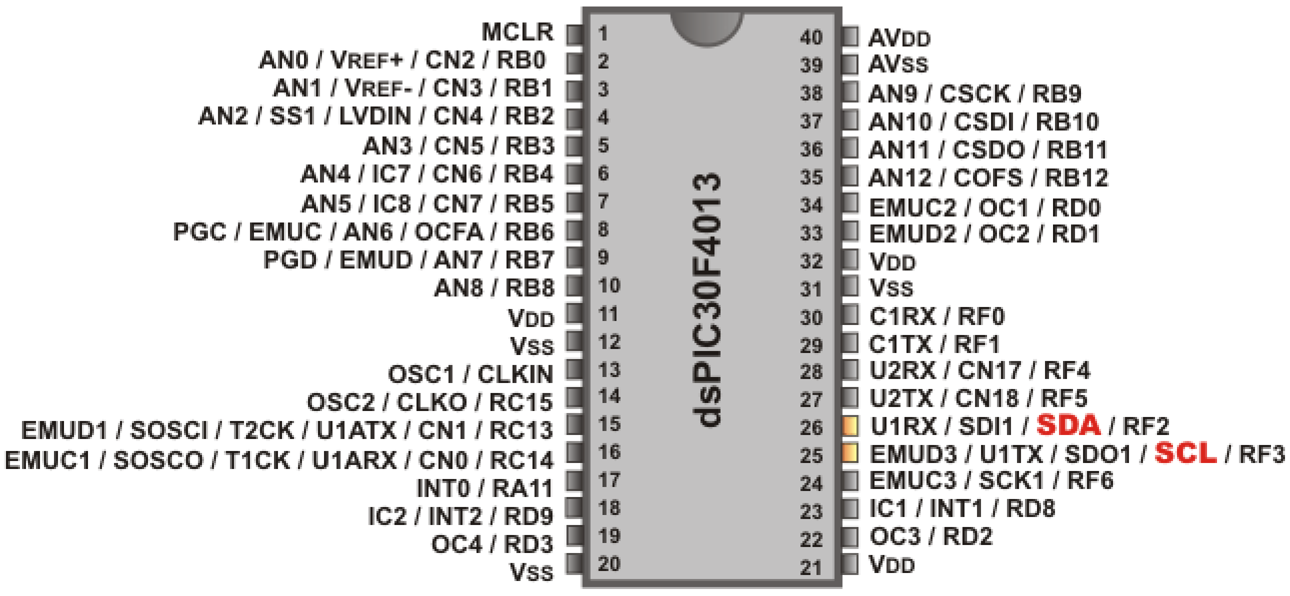


Fig. 12-3a Pinout of the dsPIC30F4013 device

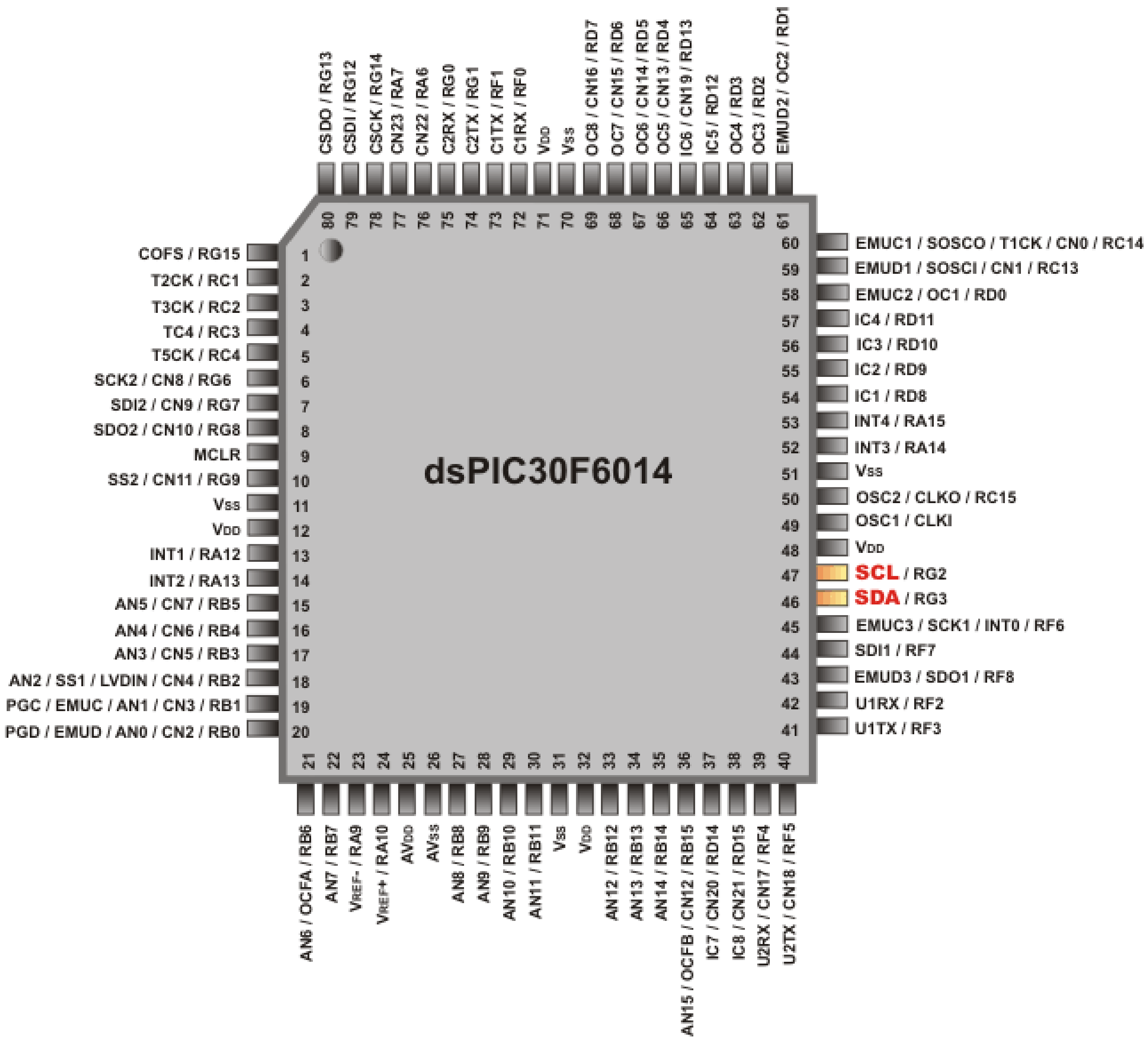


Fig. 12-3b Pinout of the dsPIC30F6014A device

Finally, a description of the I2C module registers of the dsPIC30F4013 device is presented.

name	ADR	15	14	13	12	11	10	9
I2CRCV	0x0200	-	-	-	-	-	-	-
I2CTRN	0x0202	-	-	-	-	-	-	-
I2CBRG	0x0204	-	-	-	-	-	-	-
I2CCON	0x0206	I2CEN	-	I2CSIDL	SCLREL	IPMIEN	A10M	DISSLW
I2CSTAT	0x0208	ACKSTAT	TRSTAT	-	-	-	-	GCSTAT

Table 12-2 Description of I2C registers

8	7	6	5	4	3	2	1	0	Reset State
-	Receive buffer register								0x0000
-	Transmit buffer register								0x00FF
Baud-rate generator preset register									0x0000
SMEN	GCEN	STREN	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0x0000
ADD10	IWCOL	I2COV	D_A	P	S	R_W	RBF	TBF	0x0000
Address register									

(Table 12-2 continued)

I2CEN – I2C enable bit
(I2CEN = 0 I2C module disabled, I2CEN = 1 I2C module enabled and the SDA
and SCL pins configured as serial port pins)

I2CSIDL – STOP in IDLE mode bit (I2CSIDL = 0 continue module operation,
I2CSIDL = 1 discontinue module operation)

SCREL – SCL pin release control bit (SCREL = 0 hold SCL clock low
(clock stretch in slave mode), SCREL = 1 release SCL clock)

IPMIEN – Intelligent peripheral management interface enable bit
(IPMIEN = 0 IPMI mode not enabled, IPMIEN = 1 enable IPMI support mode)

A10M – 10-bit slave address bit
(A10M = 0 I2CADD is a 7-bit address, I2CADD = 1 I2CADD is a 10-bit address)

DISSLW – Disable slew rate control bit
(DISSLW = 0 slew rate control enabled, DISSLW = 1 slew rate control disabled)

SMEN – SMBus input levels bit
(SMEN = 0 disable SMBus input thresholds, SMEN = 1 enable I/O pin
thresholds compliant with SMBus specification)

GCEN – General call enable bit (when operating as I2C slave)
(GCEN = 0 general call address disabled, GCEN = 1 enable
interrupt when a general call address is received in the I2CRSR)

STREN – SCL clock stretch enable bit (when operating as I2C slave)
(STREN = 0 disable software or receive clock stretching,
STREN = 1 enable software or receive clock stretching)

ACKDT – Acknowledge data bit (when operating as I2C master, during master receive)
(ACKDT = 0 send NACK during Acknowledge, ACKDT = 1 send ACK during Acknowledge)

ACKEN – Acknowledge sequence enable bit (when operating as I2C master, during master receive)
(ACKEN = 0 acknowledge sequence not in progress, ACKEN = 1 initiate acknowledge
sequence on SDA and SCL pins and transmit ACKDT data bit)

RCEN – Receive enable bit (when operating as I2C master)
(RCEN = 0 receive sequence not in progress, RCEN = 1 enable receive mode for I2C)

PEN – STOP condition enable bit (when operating as I2C master)
(PEN = 0 STOP condition not in progress,
PEN = 1 initiate STOP condition on SDA and SCL pins)

RSEN – Repeated START condition enable bit (when operating as I2C master)
(RSEN = 0 repeated START condition not in progress,
RSEN = 1 initiate repeated START condition on SDA and SCL pins)

SEN – START condition enabled bit (when operating as I2C master)
(SEN = 0 START condition not in propgress,
SEN = 1 initiate START condition on SDA and SCL pins)

ACKSTAT – Acknowledge status bit (when operating as I2C master)
(ACKSTAT = 0 ACK received from slave,
ACKSTAT = 1 NACK received from slave)

TRSTAT – Transmit status bit (when operating as I2C master)

(TRSTAT = 0 master transmit is not in progress,
TRSTAT = 1 master transmit is in progress (8 bits + ACK))

BCL – Master bus collision detect bit (BCL = 0 no collision,
BCL = 1 a bus collision has been detected during a master operation)

GCSTAT – General call status bit (GCSTAT = 0 general call address was not received,
GCSTAT = 1 general call address was received)

ADD10 – 10-bit address status bit
(ADD10 = 0 10-bit address was not matched, ADD10 = 1 10-bit address was matched)

IWCOL – Write collision detect bit
(IWCOL = 0 no collision, IWCOL = 1 an attempt to write the I2CTRN register failed
because the I2C module is busy)

I2COV – Receive overflow flag bit (I2COV = 0 no overflow, I2COV = 1 a byte was received while
the I2CRCV register is still holding the previous byte)

D_A – Data/address bit (when operating as I2C slave)
(D_A = 0 indicates that the last byte received was device address,
D_A = 1 indicates that the last byte received was data)

P – STOP bit
(P = 0 STOP bit was not detected last,
P = 1 indicates that a STOP bit has been detected last)

S – START bit (S = 0 START bit was not detected last,
S = 1 indicates that a START, or repeated START, bit has been detected last)

R_W – Read/write bit information (when operating as I2C slave)
(R_W = 0 write – indicates data transfer is input to slave,
R_W = 1 read - indicates data transfer is output from slave)

RBF – Receive buffer full status bit (RBF = 0 receive not complete,
I2CRCV is empty, RBF = 1 receive complete, I2CRCV is full)

TBF – Transmit buffer full status bit (TBF = 0 transmit complete,
I2CTRN is empty, TBF = 1 transmit in progress, I2CTRN is full)

Chapter13: Examples

Example 1 – Operating alpha-numeric LCD module by using 4-bit interface

The example shows the connection of an alpha-numeric LCD module 2x16 characters to a dsPIC30F microcontroller by using a 4-bit interface. The following code demonstrates usage of the LCD Custom Library routines. The example covers the initialization of the LCD module and instructions for contolling and writing the module. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. The interconnection of the LCD module and a dsPIC30F device by using a 4-bit interface is shown in Fig. 13-1.

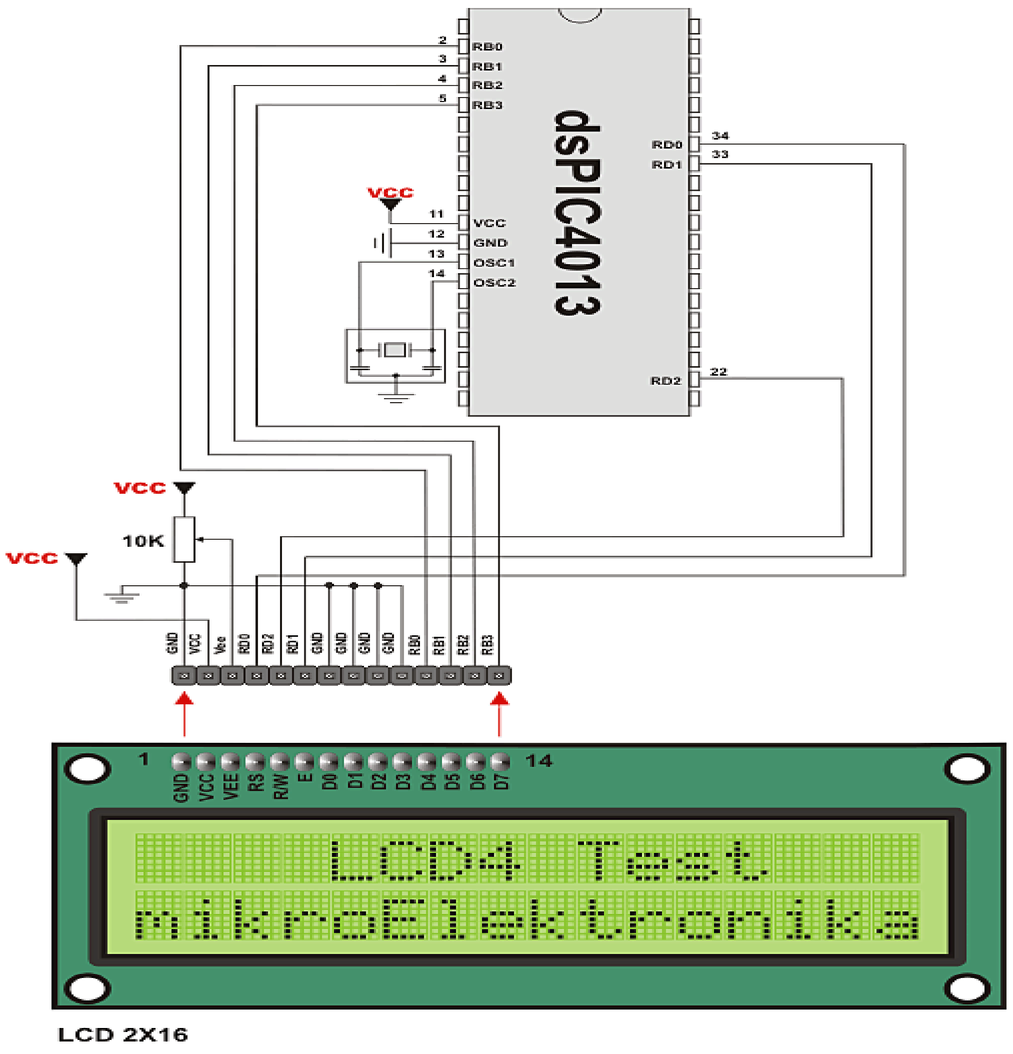


Fig. 13-1 Interconnection of the LCD module and a dsPIC30F device by using a 4-bit interface

```
char text[6] = "mikro";



void main() {



  //--- PORTB - all digital

  ADPCFG = 0xFFFF;



  Lcd_Custom_Config(&PORTB, 3,2,1,0, &PORTD, 0,2,1);

  Lcd_Custom_Out(1,3, text);

  Lcd_Custom_Out(2,6, text);

  Lcd_Custom_Chr(2,7, 'a');

  Lcd_Custom_Out(1,10, text);

  Lcd_Custom_Chr(1,11, 'o');

}//~!
```

Example 2 – Operating alpha-numeric LCD module by using 8-bit interface

The example shows the connection of an alpha-numeric LCD module 2x16 characters to a dsPIC30F microcontroller by using an 8-bit interface. The following code demonstrates usage of the LCD 8-bit Library routines. The example covers the initialization of the LCD module and instructions for contolling and writing the module. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. The interconnection of the LCD module and a dsPIC30F device by using an 8-bit interface is shown in Fig. 13-2.

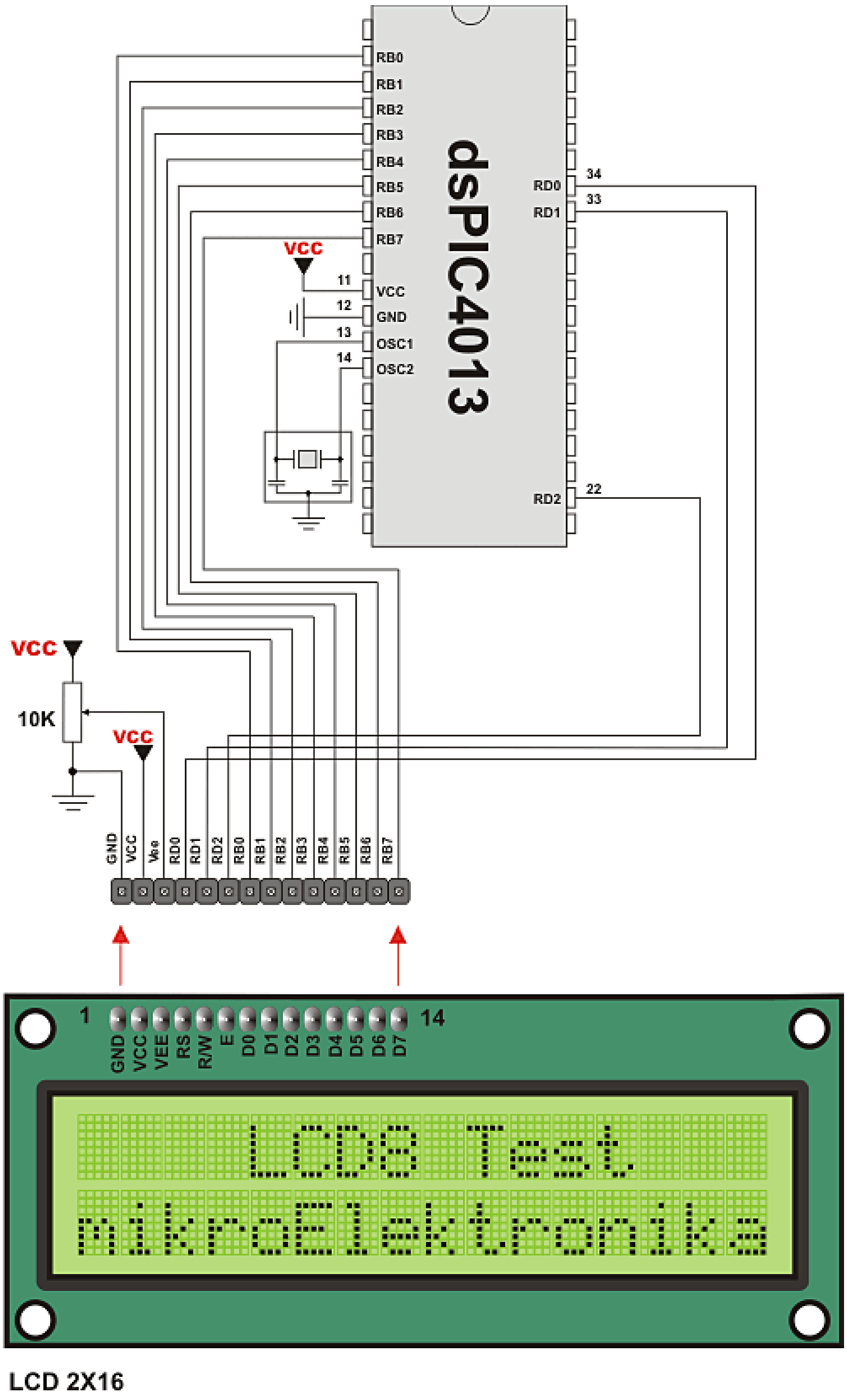
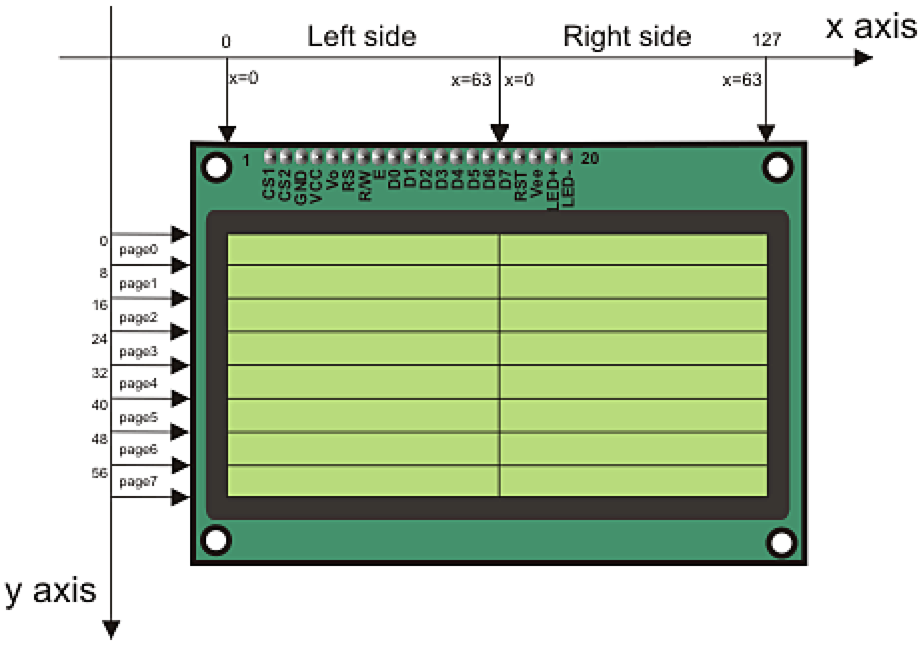


Fig. 13-2 Interconnection of the LCD module and a dsPIC30F device by using a 8-bit interface

```
void main(){



  //--- PORTB - all digital
  ADPCFG = 0xFFFF;



  Lcd8_Custom_Config(&PORTB, 7, 6, 5, 4, 3, 2, 1, 0, &PORTD, 0, 1, 2);

  Lcd8_Custom_Cmd(LCD_CLEAR);

  Lcd8_Custom_Cmd(LCD_CURSOR_OFF);



  Lcd8_Custom_Out(1, 1, "mikroElektronika");

  Lcd8_Custom_Chr(2, 1,'c');

  Lcd8_Custom_Chr_CP('?');

  Lcd8_Custom_Out_CP("for_dsPIC");

}//~!
```

Example 3 – Operating a graphical alpha-numeric LCD module (GLCD)

The example shows the connection of a «dot-matrix» graphical alpha-numeric LCD module (GLCD) to a dsPIC30F microcontroller. The example covers the initialization of the GLCD, writing text, drawing lines, boxes, and circles. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. The interconnection of the GLCD module and a dsPIC30F device is shown in Fig. 13-3.



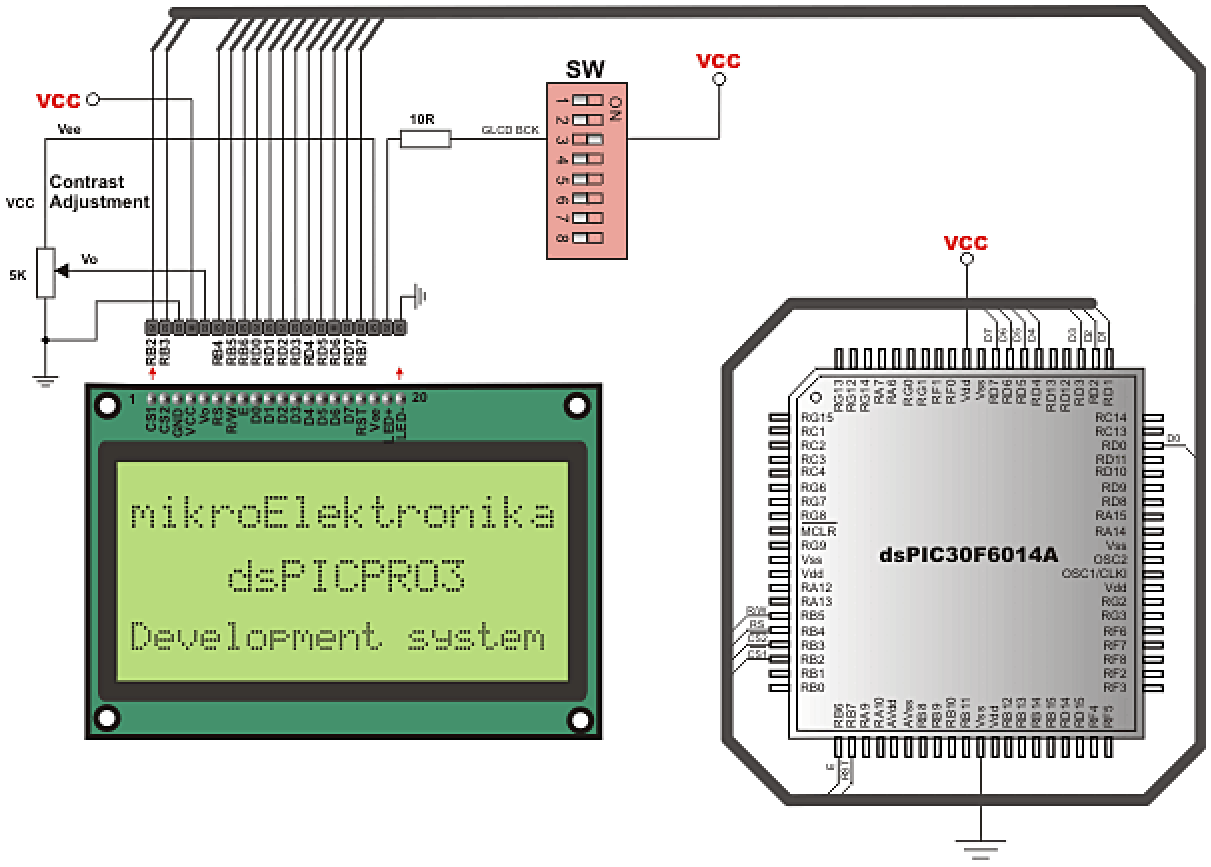


Fig. 13-3 Interconnection of the GLCD module and a dsPIC30F device.

```
#include "bmp1.h"



char cArr[20];
char *someText;



void Delay2S(){

  delay_ms(2000);

}//~





void main() {

  unsigned short ii;

  unsigned int jj;





  sometext = cArr;



  //--- turn off A/D inputs

  ADPCFG = 0xFFFF;

  // Init for dsPICPRO3 development system

  Glcd_Init(&PORTB, 2, &PORTB,3, &PORTB,4, &PORTB,5, &PORTB,7, &PORTB,6,
&PORTD);

  Delay_100ms();



  lMainLoop:
  Glcd_Fill(0x00);

  Glcd_Image( maska_bmp );

  Delay2S();



  Glcd_Fill(0x00);

  Glcd_Circle(63,32, 20, 1);

  Delay2S();

  Glcd_Line(120,1, 5,60, 1);

  Glcd_Line(12,42, 5,60, 1);

  Delay2S();



  Glcd_Rectangle(12,20, 93,57, 1);

  Delay2S();



  Glcd_Line(120,12, 12,60, 1);

  Delay2S();



  Glcd_H_Line(5,15, 6, 1);

  Glcd_Line(0,12, 120,60, 1);

  Glcd_V_Line(7,63, 127, 1);

  Delay2S();
```

```
for (ii = 1; ii <= 10; ii++)

  Glcd_Circle(63,32, 3*ii, 1);

Delay2S();



Glcd_Box(12,20, 70,57, 2);

Delay2S();



Glcd_Set_Font(defaultFont, 5,7, 48);

someText = "BIG:ONE";

Glcd_Write_Text(someText, 5,3, 2);

Delay2S();





someText = "SMALL:NOT:SMALLER";

Glcd_Write_Text(someText, 20,5, 1);

Delay2S();



Glcd_Fill(0x00);

Glcd_Set_Font(System3x6, 3, 6, 0x20);

Glcd_Write_Text(someText, 10,5, 1);

Delay2S();
```

```
  goto lMainLoop;

}//~!
```

Example 4 – Operating an AD converter

The example shows sampling by an AD converter and sends it as a text via UART1. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. Fig. 13-4 shows the connection of a dsPIC30F6014A microcontroller for sampling the voltages of the sliding contact of a potentiometer connected to the pin AN10 and sending it as a text via UART1.

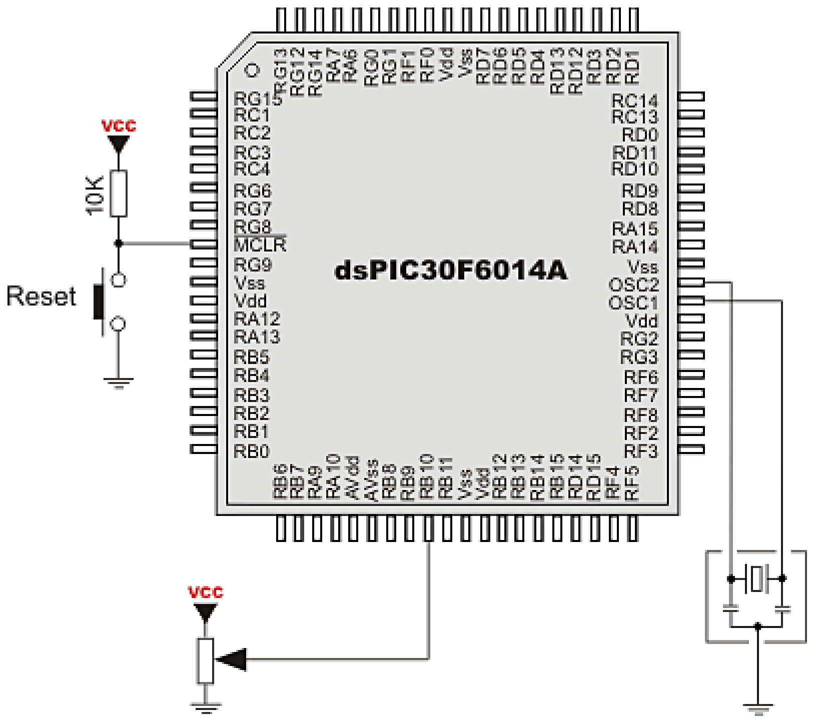

vcc
10K
Reset
vcc
dsPIC30F6014A
RG13 RG12 RG14 RA7 RA6 RG0 RG1 RF1 RF0 Vdd Vss RD7 RD6 RD5 RD4 RD13 RD12 RD3 RD2 RD1
RG15 RC1 RC2 RC3 RC4 RG6 RG7 RG8 MCLR RG9 Vss Vdd RA12 RA13 RB5 RB4 RB3 RB2 RB1 RB0
RC14 RC13 RD0 RD11 RD10 RD9 RD8 RA15 RA14 Vss OSC2 OSC1 Vdd RG2 RG3 RF6 RF7 RF8 RF2 RF3
RB6 RB7 RA9 RA10 AVdd AVss RB8 RB9 RB10 RB11 Vss Vdd RB12 RB13 RB14 RB15 RD14 RD15 RF4 RF5

Fig. 13-4 Connection of a dsPIC30F6014A device in order to sample on pin AN10

```
unsigned adcRes;

char txt[6];



void Uart1_Write_Text(char *txt_to_wr) {

  while (*txt_to_wr)

  Uart1_Write_Char(*(txt_to_wr++));

}
```

```
void main() {

  TRISBbits.TRISB10 = 1; // set pin as input - needed for ADC to work

  Uart1_Init(9600);

  while (1) {
    adcRes = Adc_Read(10);
    WordToStr(adcRes, txt);
    Uart1_Write_Text(txt);
    Delay_ms(50);
  }
}//~!
```

Example 5 – Operating a 4 x 4 keyboard

The example shows the connection of a 4 x 4 keyboard to a dsPIC30F6014A microcontroller. The example shows decoding the keyboard [0...15] to obtain ASCII symbols [0...9,A...F]. Also shown is a counter of the pressed keys. The value of the counter is written in the second line of an LCD module. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. Fig. 13-5 shows the interconnection of the keyboard and a dsPIC30F6014A microcontroller.

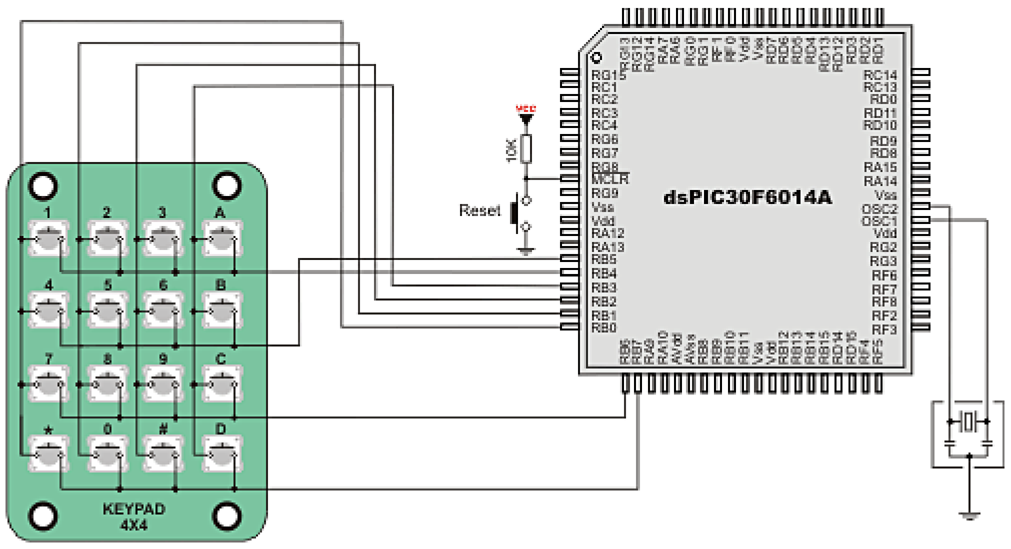


Fig. 13-5 Interconnection of the keyboard and a dsPIC30F6014A microcontroller

```
unsigned kp;



void main() {



  ADPCFG = 0xFFFF;



  Keypad_Init(&PORTB);    // PORTB [7..0]

  Uart1_Init(9600);

  Delay_ms(200);

  Uart1_Write_Char('R');



  do {

    //--- Wait for key to be pressed

    do {

      //kp = Keypad_Key_Click();     // choose the key detecting function
      kp = Keypad_Key_Press();

    } while (!kp);



    //--- Prepare value for output

    switch (kp) {





    // uncomment this block for keypad4x3 //



      /* case 10: kp = 42; break;  // '*'

         case 11: kp = 48; break;  // '0'

         case 12: kp = 35; break;  // '#'

         default: kp += 48;                      */







    // uncomment this block for keypad4x4 //



      case  1: kp = 49; break; // 1

      case  2: kp = 50; break; // 2

      case  3: kp = 51; break; // 3

      case  4: kp = 65; break; // A
```

```
        case  5: kp = 52; break; // 4

        case  6: kp = 53; break; // 5

        case  7: kp = 54; break; // 6

        case  8: kp = 66; break; // B

        case  9: kp = 55; break; // 7

        case 10: kp = 56; break; // 8

        case 11: kp = 57; break; // 9

        case 12: kp = 67; break; // C

        case 13: kp = 42; break; // *

        case 14: kp = 48; break; // 0

        case 15: kp = 35; break; // #

        case 16: kp = 68; break; // D



      }



      //--- Send on UART1

      Uart1_Write_Char(kp);



    } while (1);

  } //~!
```

Also, an example is given of the clearance by the software of the errors caused by the keyborad bounsing using the same connection as shown in Fig. 13-5. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers.

```
unsigned int oldstate;



void main() {

  ADPCFG = 0xFFFF;

  TRISB = 0xFFFF;

  TRISD = 0x0000;



  do {

    if (Button(&PORTB, 0, 1, 1))

      oldstate = 1;

    if (oldstate && Button(&PORTB, 0, 1, 0)) {

      LATD = ~LATD;

      oldstate = 0;

    }

  } while(1);

}
```

Example 6 – Realization of pulse width modulation

The example shows the realization of continuously varying pulse width modulation (PWM). The continously modulated output is on the PORTE.0 pin which is monitored by a LED diode. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. Fig. 13-6 shows the electric diagram of the realized continuous PWM of a LED diode connected to PORTE.0.

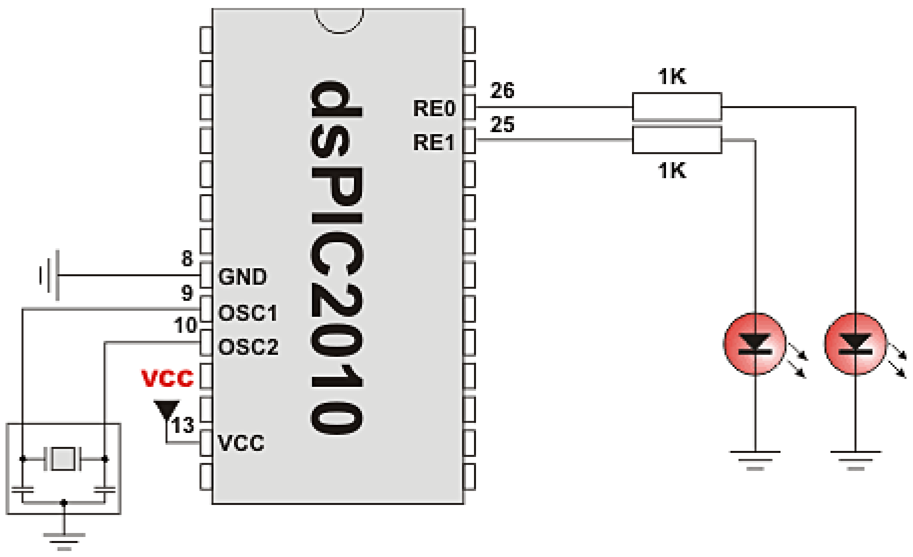


Fig. 13-6 The electric diagram of the realized continuous PWM of a LED diode connected to PORTE.0.

```
unsigned int i;



unsigned int duty_50;



void main(){

      ADPCFG = 0xFFFF;

      PORTB = 0xAAAA;

      TRISB = 0;

      Delay_ms(1000);



      duty_50 = Pwm_Mc_Init(5000,1,0x01,0);           // Pwm_Mc_Init
returns 50% of the duty

      Pwm_Mc_Set_Duty(i = duty_50,1);

      Pwm_Mc_Start();
      do

      {

        i--;

        Pwm_Mc_Set_Duty(i,1);

        Delay_ms(1);

        if (i == 0)

          i = duty_50 * 2 - 1;                    // Let us not allow
overflow

        PORTB = i;

      }

      while(1);

}//~
```

Example 7 – Operating a compact flash memory card

The example shows the connection of a compact flash (CF) memory card to a dsPIC30F micocontroller. The CF memory cards are often used as the memory elements in digital video cameras. The memory capacity is high, from 8MB up to 2GB even more, and the read/write time is typically of the order of µs. Application of CF memory cards in microcontroller systems is quite widespread.

In the CF memory cards data are split into sectors (usually of 512 bytes, in earlier models 256 bytes). Reading or writing is not performed directly byte after byte, but data are blocked per sectors through a 512 byte buffer. Fig. 13-7 shows the electrical connection of a CF memory card to a device of the dsPIC30F family. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. The example covers, writing and reading one byte from a CF memory card.

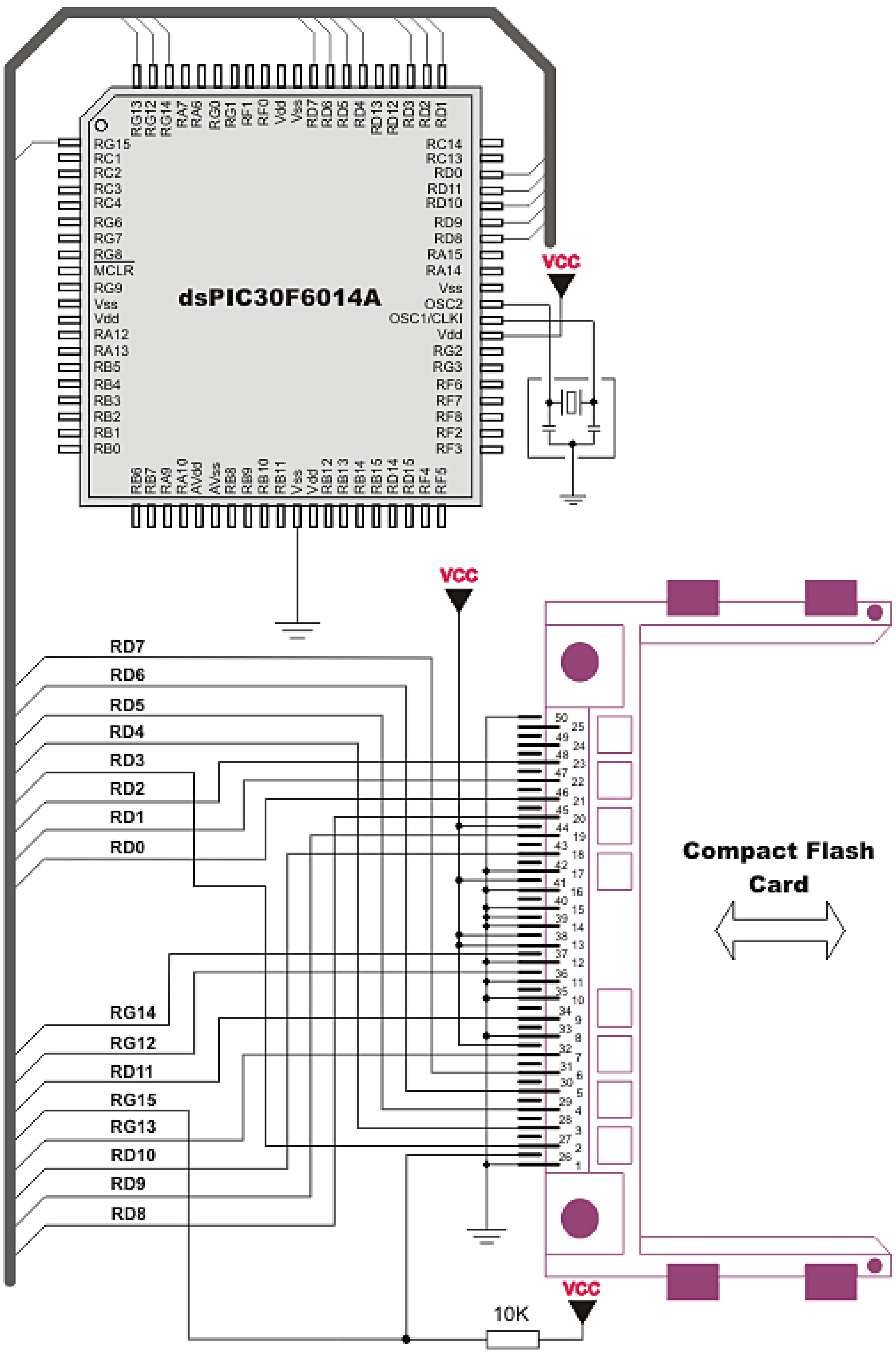


Fig. 13-7 Electrical connection of a CF memory card to a device of the dsPIC30F family.

```
char buff[512];



//-------------- Init for dsPICPRO2

void Cf_Init_dsPICPRO2() {
  Cf_Init(&PORTD,8,9,10, &PORTG,14, &PORTG,12, &PORTD,11, &PORTG,15,
&PORTG,13, &PORTD);



}//~





void initCF() {

  ADPCFG = 0xFFFF;

  Cf_Init_dsPICPRO2();



  //--- CD1 does not work on non-TTL inputs

  //while (CF_Detect() == 0) ;              // wait until CF card is
inserted

  //Delay_ms(500);                          // wait for a while until the
card is stabilized

}//~





//--------------

void testBytes() {

  unsigned int i, tmp;



  //--- write some data

  CF_Write_Init(620,1);                   // Initialize writing at sector
address 620
```

```
                                        //     for 1 sector (byte)

  Uart1_Write_Char('s');                // Notify that writing has
started

  Delay_ms(1000);

  for (i=0; i<=511; i++) {              // Write 512 bytes to sector 590

    CF_Write_Byte(i);

  }

  Delay_ms(1000);



  //--- read written data

  CF_Read_Init(620,1);                  // Initialize read from sector
address 620

  Delay_ms(1000);

                                        //    for 1 sector (byte)

  Cf_Read_Sector(620, buff);

  for (i=0; i<=511; i++) {              // Read 512 bytes from
initialized sector

    Uart1_Write_Char(buff[i]);

 }

}//~





//-------------- Main program
```

```
void main() {

  Uart1_Init(19200);



  initCF();

  testBytes();

}//~!
```

Example 8 – Operating UART modules

The example shows the initialization, writing, and reading data from the transmitter and receiver of an UART module, respectively. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. Fig. 13-8 shows the electrical connection of an UART module to an RS-232 transiever and further connection to the serial port of a PC.

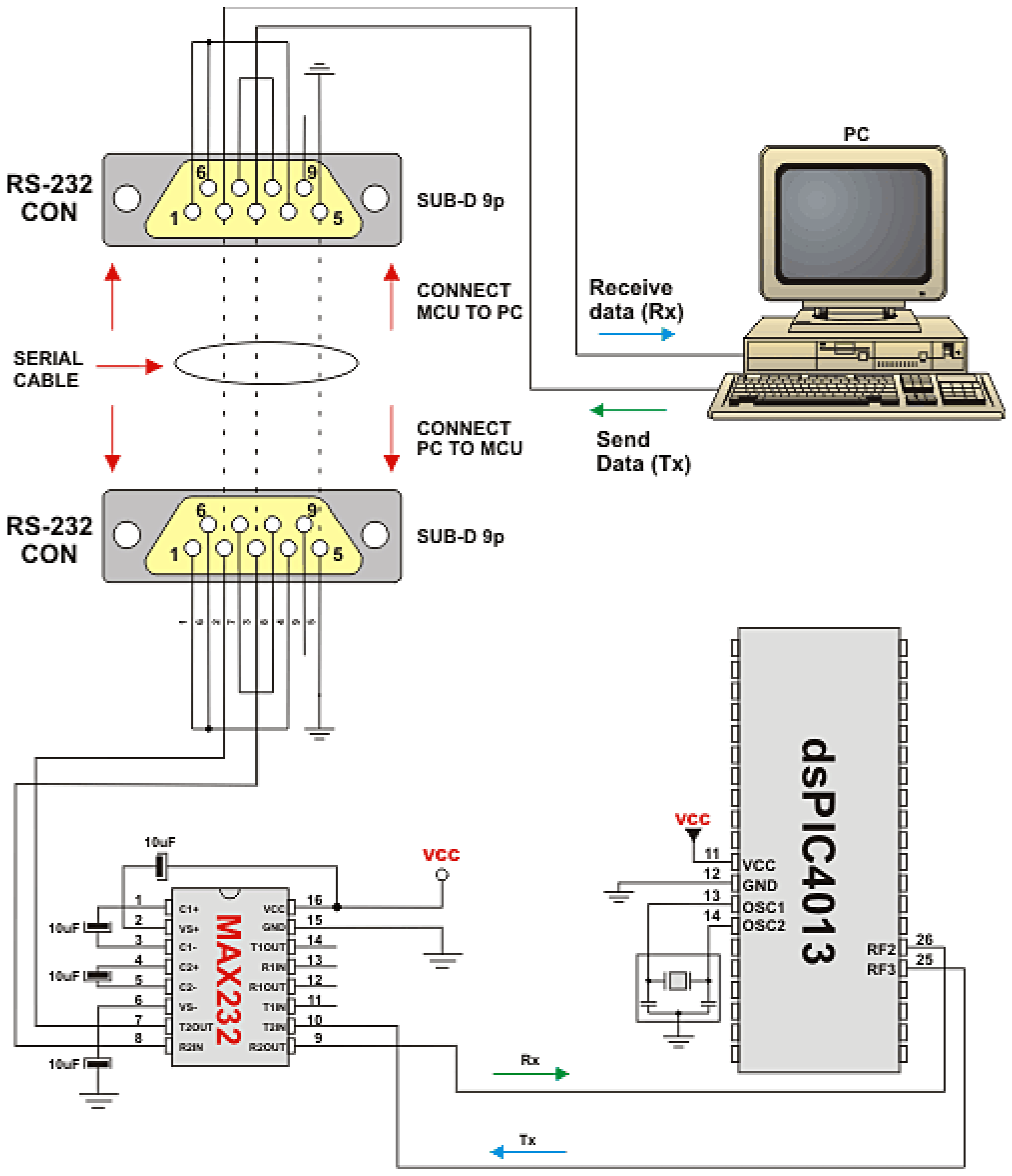


Fig. 13-8 Electrical connection of an UART module to an RS-232 transiever and further connection to the serial port of a PC.

```
unsigned rx1;



void main() {



  Uart1_Init(9600);

  Uart1_Write_Char('s');



  while(1)

  {

    if (Uart1_Data_Ready())  {

     rx1 = Uart1_Read_Char();

     Uart1_Write_Char(rx1);

    }

  }

}
```

Example 9 – Operating SPI modules

The example shows the initialization, writing, and reading data from the receive and transmit buffer register of an SPI module, respectively. The example shows the connection of the SPI2 module to the serial digital-to-analogue converter (DAC) MCP4921. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. Fig. 13-9 shows the electrical connection of the SPI module to the serial DAC MCP4921.

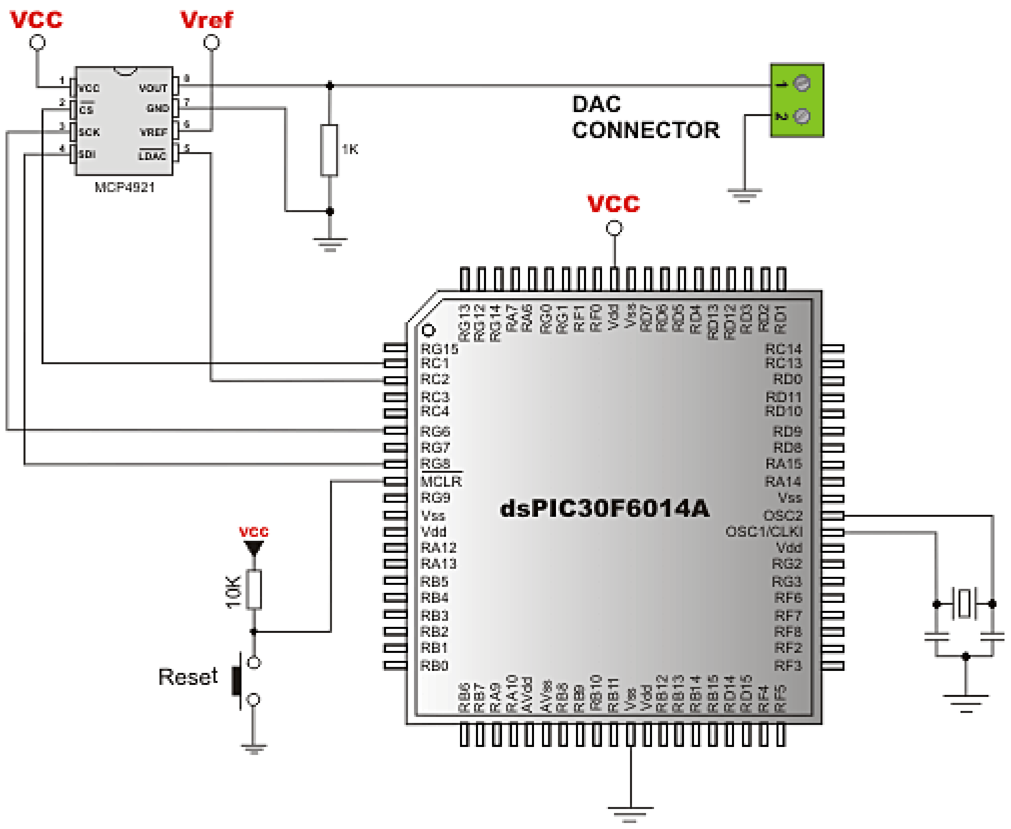


Fig. 13-9 Electrical connection of the SPI module to the serial DAC MCP4921.

```
const char CS_PIN = 1;

const char LD_PIN = 2;



unsigned int value;



void InitMain() {

  ADPCFG = 0xFFFF;                        // Set AN pins as digital



  Spi2_Init();                            // Initialize SPI2 module



  TRISC.CS_PIN = 0;                       // Set CS pin as output

  TRISC.LD_PIN = 0;                       // Set LD pin as output
```

```
}//~




// DAC increments (0..4095) --> output voltage (0..Vref)

void DAC_Output(unsigned int valueDAC) {

 char temp;



  PORTC.CS_PIN = 0;                         // Select DAC module

  PORTC.LD_PIN = 0;                         // Enable data transfer



  // Send 2 bytes of valueDAC variable

  temp = (valueDAC >> 8) & 0x0F;            // Prepare hi-byte for transfer

                                            // It's a 12-bit number, so only

                                            // lower nibble of high byte is
used

  temp |= 0x30;                             // Set MCP4921 control bits

  Spi2_Write(temp);                         // Send data via SPI



  temp = valueDAC;                          // Prepare lo-byte for transfer

  Spi2_Write(temp);                         // Send data via SPI



  PORTC.LD_PIN = 1;                         // Disable data transfer

  PORTC.CS_PIN = 1;                         // Deselect DAC module
```

```
}//~



void main() {

  InitMain();


  value = 2047;                           // When program starts, DAC gives

                                          // the output in the mid-range


  while (1) {                             // Main loop

    DAC_Output(value++);

    if (value > 4095)

      value = 0;

    Delay_ms(5);

  }

}//~!
```

Example 10 – Operating Secure Digital memory cards

Secure Digital (SD) is a standard flash memory card based on the earlier Multi Media Card (MMC) standard. Similarly to the CF memory cards, the SD memory cards have a high memory capacity, up to 2GB, and have found the application in the mobile phones, MP3 players, digital cameras, and PDA computers. Access to an SD memory card is realized by using the SPI communication interface.

This example consists of several blocks that demonstrat various aspects of usage of the Mmc_Fat16 library. These are: Creation of new file and writing down to it; Opening existing file and re-writing it (writing from start-of-file); Opening existing file and appending data to it (writing from end-of-file); Opening a file and reading data from it (sending it to USART terminal); Creating and modifying several files at once; Reading file contents; Deleting file(s); Creating the swap file (see Help for details);

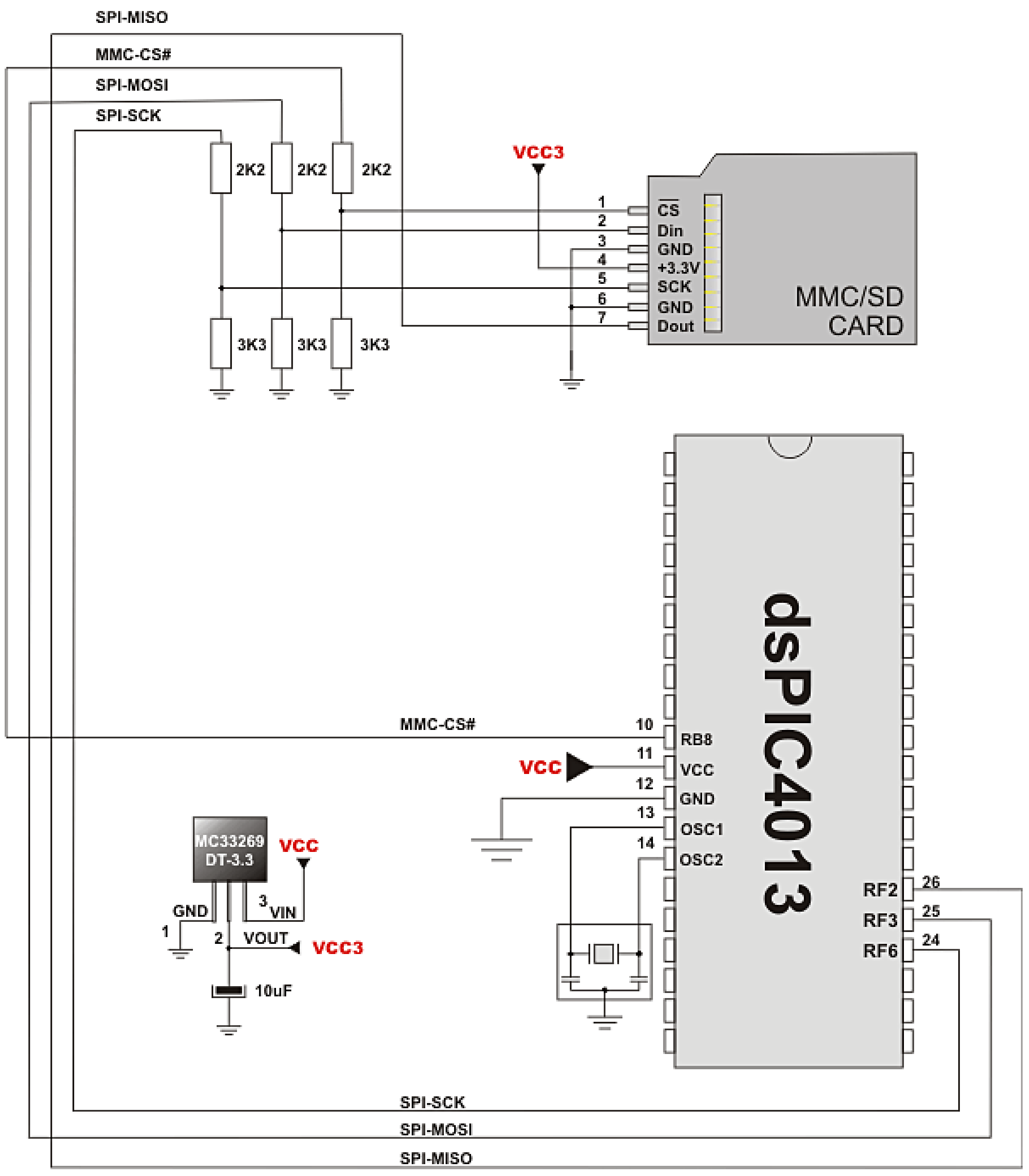


Fig. 13-10 Electrical connection of the SD memory card to a device from the dsPIC30F family.

```
#include <spi_const.h>



const char SWAP_FILE_MSG[] = "Swap file at: ";



char

 fat_txt[20] = "FAT16 not found",
 file_contents[50] = "XX MMC/SD FAT16 library by Anton Rieckertn";


char

 filename[14] = "MIKRO00xTXT";          // File names

unsigned

 loop, loop2;

unsigned short

 caracter;

unsigned long

 i, size;

char Buffer[512];


//I-I-I--------- Writes string to USART

void I_Write_Str(char *ostr) {

  unsigned i;


  i = 0;

  while (ostr[i]) {

    Uart1_Write_Char(ostr[i++]);

  }

}//~
```

```
//M-M-M--------- Creates a new file and writes some data to it

void M_Create_New_File() {

  filename[7] = 'A';

  Mmc_Fat_Assign(&filename, 0xA0);         // Will not find file and then
create file

  Mmc_Fat_Rewrite();                    // To clear file and start with new
data

  for(loop = 1; loop <= 99; loop++) {   //  We want 5 files on the MMC card

    Uart1_Write_Char('.');

    file_contents[0] = loop / 10 + 48;

    file_contents[1] = loop % 10 + 48;

    Mmc_Fat_Write(file_contents, 42);   // write data to the assigned file

  }

}//~



//M-M-M--------- Creates many new files and writes data to them

void M_Create_Multiple_Files() {

  for(loop2 = 'B'; loop2 <= 'Z'; loop2++) {

    Uart1_Write_Char(loop2);             // signal the progress

    filename[7] = loop2;                 // set filename

    Mmc_Fat_Assign(&filename, 0xA0);        // find existing file or create
a new one

    Mmc_Fat_Rewrite();                   // To clear file and start with
new data

    for(loop = 1; loop <= 44; loop++) {
      file_contents[0] = loop / 10 + 48;

      file_contents[1] = loop % 10 + 48;

      Mmc_Fat_Write(file_contents, 42);  // write data to the assigned file

    }

  }

}//~



//M-M-M--------- Opens an existing file and rewrites it

void M_Open_File_Rewrite() {

  filename[7] = 'C';

  Mmc_Fat_Assign(&filename, 0);

  Mmc_Fat_Rewrite();

  for(loop = 1; loop <= 55; loop++) {

    file_contents[0] = loop / 10 + 64;

    file_contents[1] = loop % 10 + 64;

    Mmc_Fat_Write(file_contents, 42);    // write data to the assigned file

  }

}//~



//M-M-M--------- Opens an existing file and appends data to it

//               (and alters the date/time stamp)

void M_Open_File_Append() {
     filename[7] = 'B';

     Mmc_Fat_Assign(&filename, 0);

     Mmc_Fat_Set_File_Date(2005,6,21,10,35,0);

     Mmc_Fat_Append();                                   // Prepare file
for append

     Mmc_Fat_Write(" for mikroElektronika 2005n", 27);   // Write data to
the assigned file

}//~



//M-M-M--------- Opens an existing file, reads data from it and puts it to
USART

void M_Open_File_Read() {

  filename[7] = 'B';

  Mmc_Fat_Assign(&filename, 0);

  Mmc_Fat_Reset(&size);                  // To read file, function returns
size of file

  for (i = 1; i <= size; i++) {

    Mmc_Fat_Read(&caracter);

    Uart1_Write_Char(caracter);          // Write data to USART

  }

}//~



//M-M-M--------- Deletes a file. If the file doesn't exist, it will first
be created

//               and then deleted.

void M_Delete_File() {
  filename[7] = 'F';

  Mmc_Fat_Assign(filename, 0);

  Mmc_Fat_Delete();

}//~



//M-M-M--------- Tests whether file exists, and if so sends its creation
date

//               and file size via USART

void M_Test_File_Exist(char fLetter) {

  unsigned long fsize;

  unsigned int year;

  unsigned short month, day, hour, minute;

  unsigned char outstr[12];



  filename[7] = fLetter;

  if (Mmc_Fat_Assign(filename, 0)) {

    //--- file has been found - get its date

    Mmc_Fat_Get_File_Date(&year, &month, &day, &hour, &minute);

    WordToStr(year, outstr);

    I_Write_Str(outstr);

    ByteToStr(month, outstr);

    I_Write_Str(outstr);

    WordToStr(day, outstr);
    I_Write_Str(outstr);

    WordToStr(hour, outstr);

    I_Write_Str(outstr);

    WordToStr(minute, outstr);

    I_Write_Str(outstr);

    //--- get file size

    fsize = Mmc_Fat_Get_File_Size();

    LongToStr((signed long)fsize, outstr);

    I_Write_Str(outstr);

  }

  else {

    //--- file was not found - signal it

    Uart1_Write_Char(0x55);

    Delay_ms(1000);

    Uart1_Write_Char(0x55);

  }

}//~



//-------------- Tries to create a swap file, whose size will be at least
100

//               sectors (see Help for details)

void M_Create_Swap_File() {

  unsigned int i;
  for(i=0; i<512; i++)
    Buffer[i] = i;

  size = Mmc_Fat_Get_Swap_File(5000, "mikroE.txt", 0x20);
  // see help on this function for details

  if (size) {
    LongToStr((signed long)size, fat_txt);
    I_Write_Str(fat_txt);

    for(i=0; i<5000; i++) {
      Mmc_Write_Sector(size++, Buffer);
      Uart1_Write_Char('.');
    }
  }
}//~

//-------------- Main. Uncomment the function(s) to test the desired
operation(s)
void main() {
     //--- prepare PORTD for signalling
     PORTD = 0;
     TRISD = 0;


     ADPCFG = 0xFFFF;


     //--- set up USART for the file read


     Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1,
_SPI_PRESCALE_PRI_64,


_SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH,
_SPI_ACTIVE_2_IDLE);




     Uart_Init(19200);


     U1MODEbits.ALTIO = 1;               // clear the way for SPI


     Delay_ms(200);                      // wait for the UART module to
stabilize


     //--- init the FAT library


     if (!Mmc_Fat_Init(&PORTB,8)) {


         // reinitialize spi at higher speed


         Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1,
_SPI_PRESCALE_PRI_4,


_SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH,
_SPI_ACTIVE_2_IDLE);


         //--- Test start


         PORTD = 0x0005;




         //--- Test routines. Uncomment them one-by-one to test certain
features


         M_Create_New_File();


         M_Create_Multiple_Files();
```

```
        M_Open_File_Rewrite();

        M_Open_File_Append();

        M_Delete_File();

        M_Create_Swap_File();



        M_Open_File_Read();

        M_Test_File_Exist('F');         // this file will not exist here

        M_Test_File_Exist('B');         // this file will exist here



    }

     else {

       I_Write_Str(fat_txt);

     }

     //--- Test termination

     PORTD = 0xFFFF;

}//~!
```

Example 11 – Operating I2C modules

The example shows the initialization, writing, and reading data from the transmit and receive buffer register ofan I2C module, respectively. The example shows the connection of an I2C module to the serial EEPROM memory 24C02. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers. Fig. 13-11 shows the electrical connection of an I2C module to the EEPROM memory 24C02. The example covers writing to the EEPROM memory, reading data, and data transfer to the PORTF.

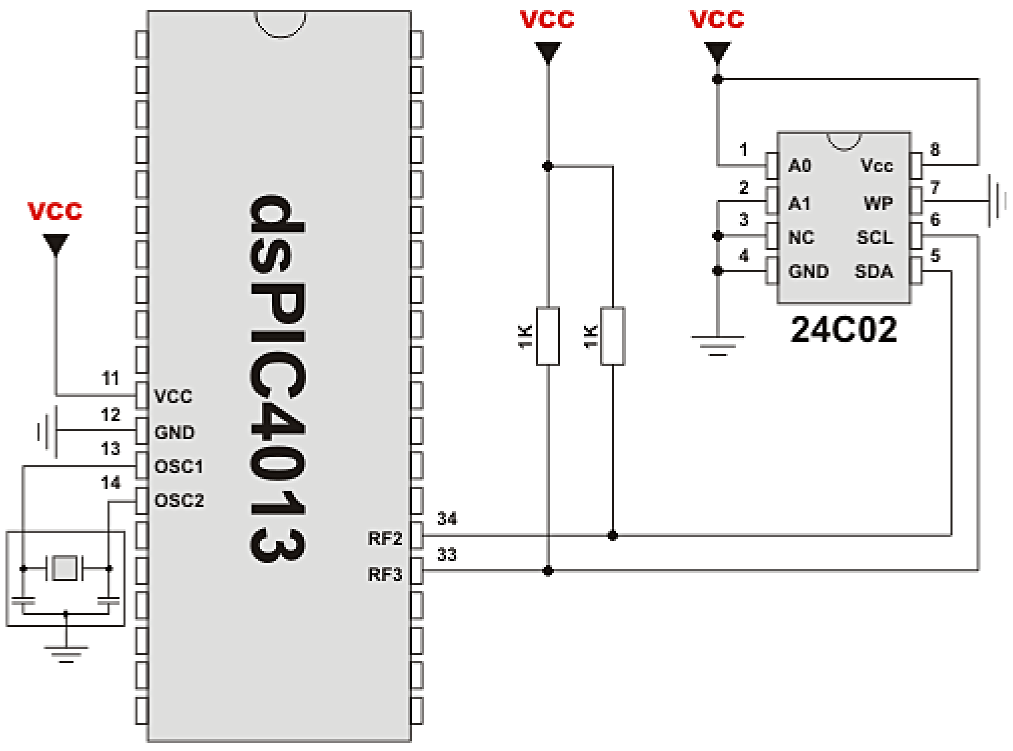


Fig. 13-11 Electrical connection of an I2C module to the EEPROM memory 24C02.

```
void main() {

  ADPCFG = 0xFFFF;

  PORTB = 0;

  TRISB = 0;



  dAddr = 0x02;



  I2c_Init(100000);

  I2c_Start();                   // issue I2C start signal

  I2c_Write(0xA2);               // send byte via I2C  (command to 24cO2)

  I2c_Write(dAddr);              // send byte (address of EEPROM location)

  I2c_Write(0xF4);               // send data (data to be written)

  I2c_Stop();
  Delay_100ms();



  I2c_Start();                      // issue I2C start signal

  I2c_Write(0xA2);                  // send byte via I2C  (device address + W)

  I2c_Write(0x02);                  // send byte (data address)

  I2c_Restart();                    // issue I2C signal repeated start

  I2c_Write(0xA3);                  // send byte (device address + R)

  PORTB = I2c_Read(1);              // Read the data (NO acknowledge)

  I2c_Stop();



}//~!
```

Also, an example is presented showing the connection of a device from the dsPIC30F family to the RTC (Real-Time Clock) generator Philips PCF8583P. The example covers writing to and reading from the RTC generator by using an I2C standard interface. Date and time are printed at LCD.

```
//RTCWrite



void main() {

   ADPCFG = 0xFFFF;

   I2C_Init(10000);         // initialize full master mode

   I2C_Start();             // issue start signal

   I2C_Write(0xA0);         // address PCF8583
   I2C_Write(0);            // start from word at address 0 (configuration
word)

   I2C_Write(0x80);         // write 0x80 to config. (pause counter...)

   I2C_Write(0);            // write 0 to cents word

   I2C_Write(0);            // write 0 to seconds word

   I2C_Write(0x30);         // write 0x30 to minutes word

   I2C_Write(0x11);         // write 0x11 to hours word

   I2C_Write(0x24);         // write 0x24 to year/date word

   I2C_Write(0x08);         // write 0x08 to weekday/month

   I2C_Stop();              // issue stop signal



   I2C_Start();             // issue start signal

   I2C_Write(0xA0);         // address PCF8530

   I2C_Write(0);            // start from word at address 0

   I2C_Write(0);            // write 0 to config word (enable counting)

   I2C_Stop();              // issue stop signal

}//~!



//RTCRead



unsigned char sec, min1, hr, day, mn, year;

char *txt, tnum[4];
```

```
void Zero_Fill(char *value) {    // fill text repesentation

  if (value[1] == 0) {           //      with leading zero

    value[1] = value[0];

    value[0] = 48;

    value[2] = 0;

  }

}//~



//--------------------- Reads time and date information from RTC (PCF8583)

void Read_Time(char *sec, char *min, char *hr, char *day, char *mn, char
*year) {

  I2C_Start();

  I2C_Write(0xA0);

  I2C_Write(2);

  I2c_Restart();

  I2C_Write(0xA1);

  *sec =I2c_Read(0);

  *min =I2c_Read(0);

  *hr =I2c_Read(0);

  *day =I2c_Read(0);

  *mn =I2c_Read(1);

  I2C_Stop();

}//~
```

```c
//-------------------- Formats date and time

void Transform_Time(char  *sec, char *min, char *hr, char *day, char *mn,
char *year) {

  *sec  =  ((*sec & 0xF0) >> 4)*10 + (*sec & 0x0F);

  *min  =  ((*min & 0xF0) >> 4)*10 + (*min & 0x0F);

  *hr   =  ((*hr & 0xF0) >> 4)*10 + (*hr & 0x0F);

  *year =  (*day & 0xC0) >> 6;

  *day  =  ((*day & 0x30) >> 4)*10 + (*day & 0x0F);

  *mn   =  ((*mn & 0x10) >> 4)*10 + (*mn & 0x0F);

}//~



//-------------------- Output values to LCD

void Display_Time(char sec, char min, char hr, char day, char mn, char
year) {

   ByteToStr(day,tnum);

   txt = rtrim(tnum);

   Zero_Fill(txt);

   Lcd_Custom_Out(1,6,txt);

   ByteToStr(mn,tnum);

   txt = rtrim(tnum);

   Zero_Fill(txt);

   Lcd_Custom_Out(1,9,txt);

   Lcd_Custom_Chr(1,15,52+year);
   ByteToStr(hr,tnum);

   txt = rtrim(tnum);

   Zero_Fill(txt);

   Lcd_Custom_Out(2,6,txt);

   ByteToStr(min,tnum);

   txt = rtrim(tnum);

   Zero_Fill(txt);

   Lcd_Custom_Out(2,9,txt);

   ByteToStr(sec,tnum);

   txt = rtrim(tnum);

   Zero_Fill(txt);

   Lcd_Custom_Out(2,12,txt);

}//~



//------------------- Performs project-wide init

void Init_Main() {

  ADPCFG = 0xFFFF;

  Lcd_Custom_Init_EasyDsPIC4();

  I2C_Init(100000);         // initialize I2C

  txt = "Date:";            // prepare and output static text on LCD

  Lcd_Custom_Out(1,1,txt);

  Lcd_Custom_Chr(1,8,':');
  Lcd_Custom_Chr(1,11,':');

  txt = "Time:";

  Lcd_Custom_Out(2,1,txt);

  Lcd_Custom_Chr(2,8,':');

  Lcd_Custom_Chr(2,11,':');

  txt = "200";

  Lcd_Custom_Out(1,12,txt);

  Lcd_Custom_Cmd(LCD_CURSOR_OFF);   // cursor off

}//~



//----------------- Main function

void main() {

  Init_Main();                                      // perform
initialization

  while (1) {

    Read_Time(&sec,&min1,&hr,&day,&mn,&year);       // read time from
RTC(PCF8583)

    Transform_Time(&sec,&min1,&hr,&day,&mn,&year);  // format date and time

    Display_Time(sec, min1, hr, day, mn, year);     // prepare and display
on LCD

    Delay_ms(1000);                                 // wait 1s

  }

}
```

Example 12 – Operating a PS/2 keyboard

The example shows the connection of a device from the dsPIC30F family to a standard PS/2 keyboard. It is important to note that all pins of the PS/2 keyboard connected to the dsPIC30F device are connected to the power supply by the pull-up resistors. The realization is carried out by using the mikroC compiler for dsPIC30F microcontrollers.

```
unsigned int

 keydata = 0, special = 0, down = 0;



void main() {

  ADPCFG = 0xFFFF;

  Uart1_Init(9600);

  Ps2_Init(&PORTC);         // Init PS/2 Keyboard on PORTC

                            // pin 13 is connected to Data line

                            // pin 14 is connected to Clock line



  Delay_ms(100);            // Wait for keyboard to finish

  Uart1_Write_Char('s'); Uart1_Write_Char('t'); Uart1_Write_Char('a');

  Uart1_Write_Char('r'); Uart1_Write_Char('t'); Uart1_Write_Char('!');

  do {

    if(Ps2_Key_Read(&keydata, &special, &down)) {

      if(down && (keydata == 16)) {// Backspace

        Uart1_Write_Char(0x08);

      }

      else if(down && (keydata == 13)) {// Enter
```

```
        Uart1_Write_Char('r');   // send carriage return to usart terminal

        //Uart1_Write_Char('n');//uncomment this line if usart terminal
also expects line feed

                                 // for new line transition

      }

      else if(down && !special && keydata) {

        //Uart1_Write_Char(keydata >> 8);

        Uart1_Write_Char(keydata);

      }

    }

    Delay_ms(1);            // debounce

  } while(1);

}//~
```

Chapter14: DSP Examples

14.1 Useful functions

At the beginning of this chapter the functions used in this chapter will be described together with some other useful functions. Table 14-1 presents a list of the functions including a description of their parameters, results (functions only), and eventual restrictions on the parameter values.

- FIR_Radix
- IIR_Radix
- FFT
- IFFT
- BitReverseComplex
- Vector_Set
- VectorPower
- Vector_Subtract
- VectorScale
- Vector_Negate
- Vector_Multiply
- Vector_Min
- Vector_Max
- Vector_Dot
- Vector_Correlate
- Vector_Convolve
- Vector_Add
- Matrix_Transpose
- Matrix_Subtract
- Matrix_Scale
- Matrix_Multiply
- Matrix_Add

FIR_Radix

Prototype	`unsigned FIR_Radix(unsigned FilterOrder, const unsigned *ptrCoeffs, unsigned BuffLength, unsigned *ptrInput, unsigned Index);`
Description	This function applies FIR filter to ptrInput. Input samples must be in Y data space. `FilterOrder` is order of the filter + 1. `ptrCoeffs` is address of filter coeffitients in program memory. `BuffLength` represents number of samples `ptrInput` points to. `ptrInput` is address of input samples. `Index` index of current sample.
Returns	**sum(k=0..N-1)(coef[k]*input[N-k])** - Current sample of processed signal(B[n]) N - buffer length k - Current index

IIR_Radix

Prototype	`unsigned IIR_Radix (const int BScale, const int AScale, const signed *ptrB, const signed *ptrA, unsigned FilterOrder, unsigned *ptrInput, unsigned Input_Len, unsigned *ptrOutput, unsigned Index)`
Description	This function applies IIR filter to ptrInput. Input and output samples must be in Y data space. `AScale` A Scale factor

	`BScale` B Scale factor `ptrB` Address of B coefficients (In program memory) `ptrA` Address of A coefficients (In program memory) `FilterOrder` is order of the filter + 1. `ptrInput` is address of input samples. `Input_Len` represents number of samples `ptrInput` points to. `ptrOutput` is address of output samples. Output length is equal to Input length. `Index` index of current sample.
Returns	**y[n]=sum(k=0..N)(Acoef[k]*x[n-k]) - sum(k=1..M)(Bcoef[k]*y[n-k])**

FFT

Prototype	`void Fft(unsigned log2N, const unsigned *TwiddleFactorsAddress, unsigned *Samples);`
Description	Function applies FFT transformation to input samples, input samples must be in Y data space. `N` - buffer length (must be the power of 2). `TwiddleFactorsAddress` is address of costant array which contains complex twiddle factors.The array is expected to be in program memory. `Samples` array of input samples. Upon completion complex array of FFT samples is placed in the Samples parameter.
Returns	**F(k) = 1/N*sum_n (f(n)*WN(kn)), WN(kn) = exp[-(j*2*pi*k*n)/N]** Fn - array of complex input samples n in {0, 1,... , N-1}, and k in {0, 1,... , N-1}, with N = 2^m, m element of Z. WN - TwiddleFactors The amplitude of current FFT sample is calculated as: **F[k]=sqrt(Re[k]^2+ Im[k]^2)**
Note	Complex array of FFT samples is placed in Samples parameter. Input Samples are arranged in manner Re,Im,Re,Im... (where Im is always zero). Output samples are arranged in the same manner but Im parts are different from zero. Output samples are symmetrical (First half of output samples (index from 0 to N/2) is identical to the second half of output samples(index from N/2 to N). Input data is a complex vector such that the magnitude of the real and imaginary parts of each of its elements is less than 0.5. If greater or equal to this value the results could produce saturation. Note that the output values are scaled by a factor of 1/N, with N the length of the FFT. input is expected in natural ordering, while output is produced in bit reverse ordering.

BitReverseComplex

Prototype	`void BitReverseComplex(unsigned log2N, unsigned *ReIm)`
Description	This function does Complex (in-place) Bit Reverse re-organization. `N` - buffer length (must be the power of 2). `ReIm` - Output Sample(from FFT).
Note	Input samples must be in Y data space.

Vector_Set

Prototype	`void Vector_Set(unsigned *input, unsigned size, unsigned value);`
Description	Sets `size` elements of `input` to `value`, starting from the first element. Size must be > 0. Length of `input` is limited by available ram

VectorPower

Prototype	`unsigned VectorPower(unsigned N, unsigned *Vector);`
Description	Function returns result of power value (powVal) in radix point 1.15
Operation	**powVal = sum (srcV[n] * srcV[n])** with n in {0, 1,... , numElems-1}
Input	`N` = number of the elements in vector(s) (numElems) `Vector` = ptr to source vector (srcV)
Note	AccuA used, not restored CORCON saved, used, restored

Vector_Subtract

Prototype	`void Vector_Subtract(unsigned *dest, unsigned *v1, unsigned *v2, unsigned numElems);`
Description	This procedure does substraction of two vectors. `numElems` must be less or equal to minimum size of two vectors. `v1` - First Vector `v2` - Second Vector `dest` - Result Vector
Operation	**dstV[n] = srcV1[n] - srcV2[n]** with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

VectorScale

Prototype	`void VectorScale(unsigned N, int ScaleValue, unsigned *SrcVector, unsigned *DestVector);`
Description	This procedure does vector scaling with scale value. `N` - Buffer length `SrcVector` - original vector `DestVector` - scaled vector `ScaleValue` - Scale Value
Operation	**dstV[n] = sclVal * srcV[n]**, with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

Vector_Negate

Prototype	`void Vector_Negate(unsigned *srcVector, unsigned *DestVector, unsigned numElems);`
Description	This procedure does negation of vector. `srcVector` - Original vector `destVector` - Result vector `numElems` - Number of Elements
Operation	**dstV[n] = (-1)*srcV1[n] + 0**, 0 <= n < numElems
Note	Negate of 0x8000 is 0x7FFF. AccuA used, not restored. CORCON saved, used, restored.

Vector_Multiply

Prototype	`void Vector_Multiply(unsigned *v1, unsigned *v2, unsigned *dest, unsigned numElems);`
Description	This procedure does multiplication of two vectors.

	`numElems` must be less or equal to minimum size of two vectors. `v1` - First Vector `v2` - Second Vector `dest` - Result Vector
Operation	**dstV[n] = srcV1[n] * srcV2[n]** with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

Vector_Min

Prototype	`**unsigned** Vector_Min(**unsigned** *Vector, **unsigned** numElems, **unsigned** *MinIndex);`
Description	This function find min. value in vector. `Vector` - Original vector. `numElems` - Number of elements `MinIndex` - Index of minimum value

Operation	**minVal = min {srcV[n], n in {0, 1,...numElems-1}**
	if srcV[i] = srcV[j] = minVal, and i < j, then minIndex = j

Returns	minimum value (minVal)

Vector_Max

Prototype	`**unsigned** Vector_Max(**unsigned** *Vector, **unsigned** numElems, **unsigned** *MaxIndex);`
Description	This function find max. value in vector. `Vector` - Original vector. `numElems` - Number of elements `MaxIndex` - Index of maximum value

Operation	**maxVal = max {srcV[n], n in {0, 1,...numElems-1} }**
	if srcV[i] = srcV[j] = maxVal, and i < j, then maxIndex = j

Returns	maximum value (maxVal)

Vector_Dot

Prototype	`**unsigned** Vector_Dot(**unsigned** *v1, **unsigned** *v2, **unsigned** numElems);`
Description	Procedure calculates vector dot product. `v1` - First vector. `v2` - Second vector `numElems` - Number of elements
Operation	**dotVal = sum (srcV1[n] * srcV2[n])**, with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

Vector_Correlate

Prototype	`**void** Vector_Correlate(**unsigned** *v1, **unsigned** *v2, **unsigned** *dest, **unsigned** numElemsV1, **unsigned** numElemsV2);`
Description	Procedure calculates Vector correlation (using convolution). `v1` - First vector. `v2` - Second vector `numElemsV1` - Number of first vector elements `numElemsV2` - Number of second vector elements

	`dest` - Result vector
Operation	**r[n] = sum_(k=0:N-1){x[k]*y[k+n]}**, where: x[n] defined for 0 <= n < N, y[n] defined for 0 <= n < M, (M <= N), r[n] defined for 0 <= n < N+M-1.

Vector_Convolve

Prototype	`void Vector_Convolve(unsigned *v1, unsigned *v2, unsigned *dest, unsigned numElemsV1, unsigned numElemsV2);`
Description	Procedure calculates Vector using convolution. `v1` - First vector. `v2` - Second vector `numElemsV1` - Number of first vector elements `numElemsV2` - Number of second vector elements `dest` - Result vector

Operation	y[n] = sum_(k=0:n){x[k]*h[n-k]}, 0 <= n < M
	y[n] = sum_(k=n-M+1:n){x[k]*h[n-k]}, M <= n < N
	y[n] = sum_(k=n-M+1:N-1){x[k]*h[n-k]}, N <= n < N+M-1

Note	AccuA used, not restored. CORCON saved, used, restored.

Vector_Add

Prototype	`void Vector_Add(unsigned *dest, unsigned *v1, unsigned *v2, unsigned numElems);`
Description	Procedure calculates vector addition. `v1` - First vector. `v2` - Second vector `numElems` - Number of vector elements `dest` - Result vector
Operation	**dstV[n] = srcV1[n] + srcV2[n]**, with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

Matrix_Transponse

Prototype	`void Matrix_Transpose(unsigned * src, unsigned * dest, unsigned num_rows, unsigned num_cols);`
Description	Procedure does matrix transposition. `src` - Original matrix. `dest` - Result matrix `numRows` - Number of matrix rows `numCols` - Number of matrix columns
Operation	**dstM[i][j] = srcM[j][i]**

Matrix_Subtract

Prototype	`void Matrix_Subtract(unsigned * src1, unsigned * src2, unsigned * dest, unsigned num_rows, unsigned num_cols);`
Description	Procedure does matrix substraction. `src1` - First matrix. `src2` - Second matrix

	`dest` - Result matrix `num_rows` - Number of matrix rows `num_cols` - Number of matrix columns
Operation	**dstM[i][j] = srcM1[i][j] - srcM2[i][j]**
Note	AccuA used, not restored. AccuB used, not restored. CORCON saved, used, restored.

Matrix_Scale

Prototype	`**void** Matrix_Scale(**unsigned** scale_value, **unsigned** *src1, **unsigned** *dest, **unsigned** num_rows, **unsigned** num_cols);`
Description	Procedure does matrix scale. `ScaleValue` - Scale Value `src1` - Original matrix `dest` - Result matrix `num_rows` - Number of matrix rows `num_cols` - Number of matrix columns
Operation	**dstM[i][j] = sclVal * srcM[i][j]**
Note	AccuA used, not restored. CORCON saved, used, restored.

Matrix_Multiply

Prototype	`**void** Matrix_Multiply(**unsigned** * src1, **unsigned** * src2, **unsigned** * dest, **unsigned** numRows1, **unsigned** numCols2, **unsigned** numCols1Rows2);`
Description	Procedure does matrix multiply. `src1` - First Matrix `src2` - Second Matrix `dest` - Result Matrix `numRows1` - Number of first matrix rows `numCols2` - Number of second matrix columns `numCols1Rows2` - Number of first matrix columns and second matrix rows

Operation	**dstM[i][j] = sum_k(srcM1[i][k]*srcM2[k][j])**,
	with
	i in {0, 1, ..., numRows1-1}
	j in {0, 1, ..., numCols2-1}
	k in {0, 1, ..., numCols1Rows2-1}

Note	AccuA used, not restored. CORCON saved, used, restored.

Matrix_Add

Prototype	`**void** Matrix_Add(**unsigned** * src1, **unsigned** * src2, **unsigned** * dest, **unsigned** numRows, **unsigned** numCols);`
Description	Procedure does matrix addition. `src1` - First Matrix `src2` - Second Matrix `dest` - Result Matrix `numRows` - Number of first matrix rows `numCols` - Number of second matrix columns
Operation	**dstM[i][j] = srcM1[i][j] + srcM2[i][j]**
Note	AccuA used, not restored.

	CORCON saved, used, restored.

14.2.1 Example 1 – Menu

The example shows a method of formation of a menu with options on an LCD in 4-bit mode. Setting of bit 1 on port F means go to the next option, whereas setting of bit 0 on port F means go to the previous option. The rate of transition to the next/previous option is limited so the maximum of 4 transitions per second is allowed.

```
/* This project is designed to work with PIC P30F6014A. It has been tested


   on dsPICPRO3 development system with 10.0 MHz crystal and 8xPLL.


   It should work with any other crystal.


   Note: the maximum operating frequency for dsPIC is 120MHz.


   With minor adjustments, this example should work with any other dsPIC
MCU */




signed short menu_index;


char menu[5][7] = {"First" ,                   // Menu items


                   "Second",


                   "Third" ,


                   "Fourth",


                   "Fifth"   } absolute 0x1880;


                   // Directive absolute specifies the starting address


                   // in RAM for a variable.




void main() {


  ADPCFG = 0xFFFF;                             // Configure PORTB as
digital
```

```
  TRISF  = 0xFFFF;                             // Configure PORTF as input
(menu control)



  menu_index = 0;                              // Init menu_item[0]



  // Init LCD in 4-bit mode for dsPICPRO3 board

  Lcd_Custom_Config(&PORTD, 7,6,5,4, &PORTB, 4,0,6);

  Lcd_Custom_Cmd(LCD_CURSOR_OFF);

  Lcd_Custom_Cmd(LCD_FIRST_ROW);

  Lcd_Custom_Out(1,1,"Menu :");



  Lcd_Custom_Out(1,8,menu[menu_index]);        // Show menu element on LCD



  while (1) {                                  // endless loop

    if (PORTF.F1 == 1) {                       // Detect logical one on RF1
pin => MENU UP

      menu_index = menu_index + 1;             // Next index in menu

      if (menu_index > 4)

        menu_index = 0;                        // Circular menu

      Lcd_Custom_Out(1,8, "      ");           // Clear text

      Lcd_Custom_Out(1,8,menu[menu_index]);    // Show menu element on LCD

      Delay_ms(250);                           // No more than 4 changes
per sec

    }
    if (PORTF.F0 == 1) {                          // Detect logical one on RF0
pin => MENU DOWN

      menu_index = menu_index - 1;                // Previous index in menu

      if (menu_index < 0)

        menu_index = 4;                           // Circular menu

      Lcd_Custom_Out(1,8, "      ");              // Clear text

      Lcd_Custom_Out(1,8,menu[menu_index]);    // Show menu element on LCD

      Delay_ms(250);                              // No more than 4 changes
per sec

    }

  }

}
```

14.2.2 Example 2 – DTMFout

The example shows a method of generation of DTMF signal. The following assumptions have been adopted:

- the level of the signal of the higher frequency is 0 to 3dB higher compared to the signal of the lower frequency,
- transmission of one character lasts 90ms, the pause between two characters is 200ms,
- the sampling frequency is 20kHz,
- input data are sent to the microcontroller via the UART module at the rate of 9600 bps,

The algorithm of generating the signal is the following. On the basis of the calculated frequencies the square signal is sent first. Then, this signal is filtered by a low pass IIR filter of the third order (4 coefficients). The output signal specifications are:

- frequency deviation is less than 1.5%,
- the level of all harmonics is 20dB below the useful signal.

The algorithm is shown in Fig. 14-1.

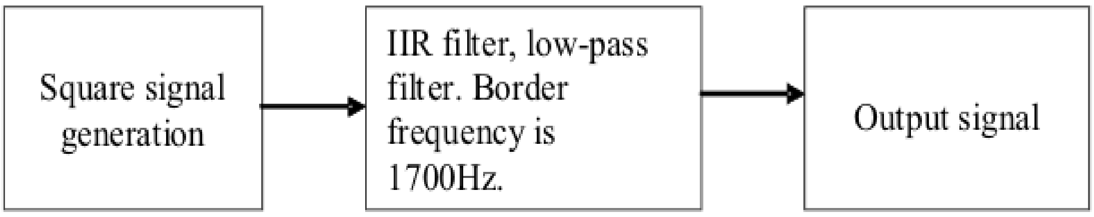


Fig. 14-1 Algorithm for generation of DTMF signal

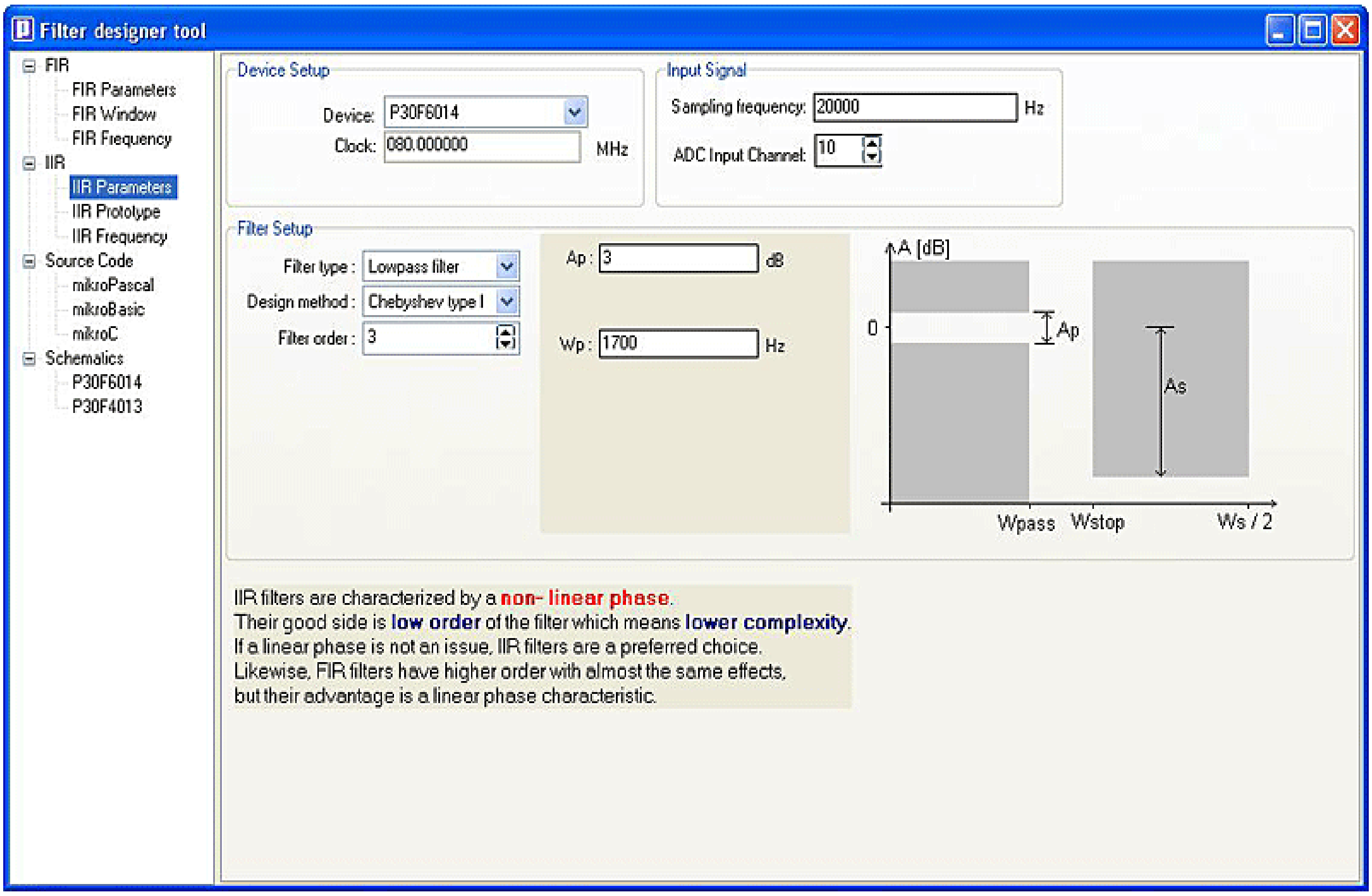
Digital filter is lowpass Chebyshev type 1 filter. Wp is 1700 Hz, Ap is 3dB, sampling frequency is 20kHz. Third order fiter can meet the requirements:

a) level of all harmonics is 20dB below the useful signal:

Harmonic nearest to the useful signal is 3xf1. f1 is lowest frequency of useful signal, f1=697 Hz. Fourier series of square wave signal implies that amplitude of 3xf1=2091 Hz harmonic is 1/3 of f1signal amplitude. So we already have -9.5dB. In IIR frequency window we can see that 3xf1 frequency is 12dB weaker than useful signal. The most critical harmonic 3xf1 is -9.5dB -12dB = -21.5dB weaker than useful signal – so this requirement is fullfilled.

b) the level of the signal of the higher frequency is 0 to 3dB higher compared to the signal of the lower frequency:

IIR frequency window also shows that low frequencies 697-941Hz filter attenuation is in -3dB to -2.8dB range and low frequencies 1209-1633Hz filter attenuation is in –1.8dB to 0dB range. So any higher frequency signal is 1 to 3dB stronger than any lower frequency signal.



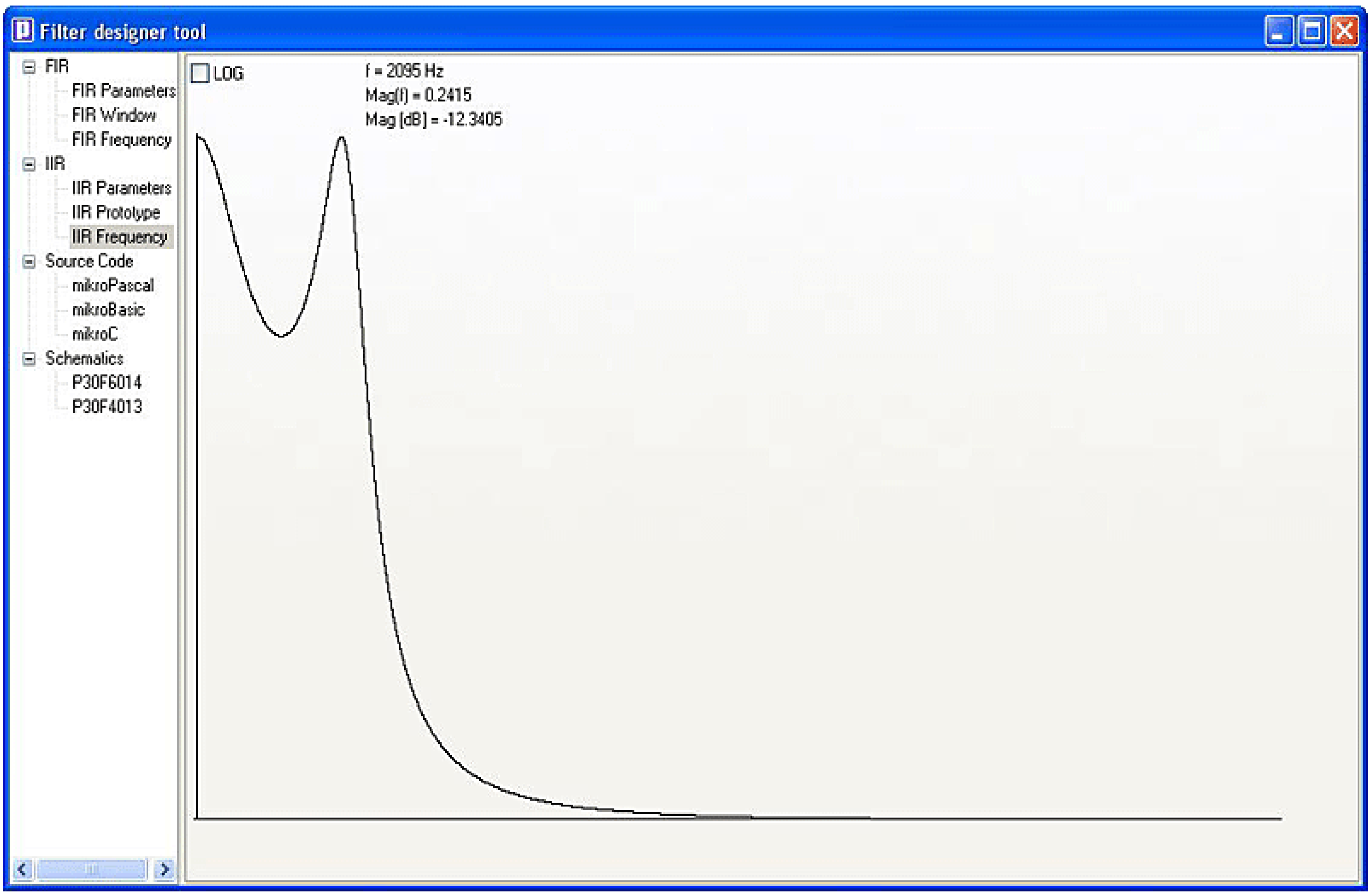
Device Setup
Device: P30F6014
Clock: 080.000000 MHz

Input Signal
Sampling frequency: 20000 Hz
ADC Input Channel: 10

Filter Setup
Filter type : Lowpass filter
Design method : Chebyshev type I
Filter order : 3
Ap : 3 dB
Wp : 1700 Hz

IIR filters are characterized by a non- linear phase.
Their good side is low order of the filter which means lower complexity.
If a linear phase is not an issue, IIR filters are a preferred choice.
Likewise, FIR filters have higher order with almost the same effects,
but their advantage is a linear phase characteristic.

Filter Designer Tool Window 1.



f = 2095 Hz
Mag(f) = 0.2415
Mag [dB] = -12.3405

Filter Designer Tool Window 2.

Fig. 14-2 shows the generated signal. The signals of the lower and higher frequencies are added, with the higher frequency signal 3dB above the lower frequency signal.

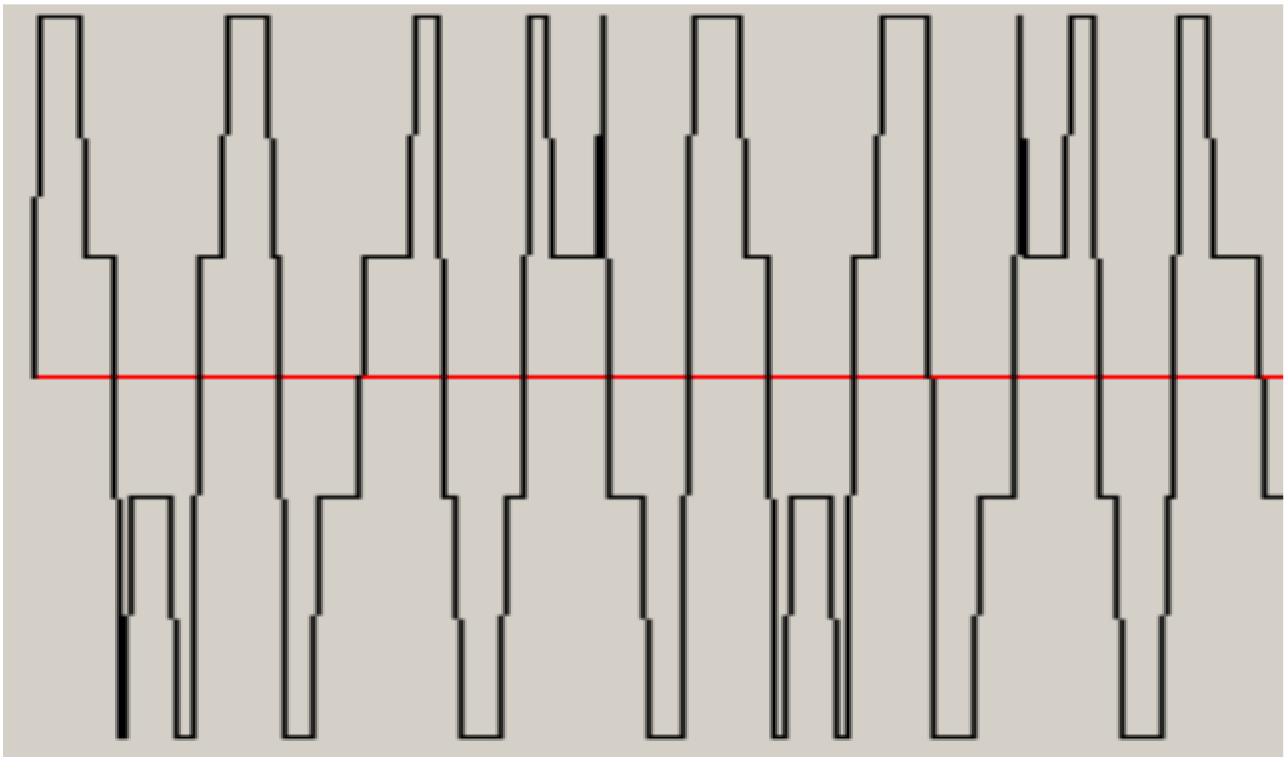


Fig. 14-2 Generated signal.

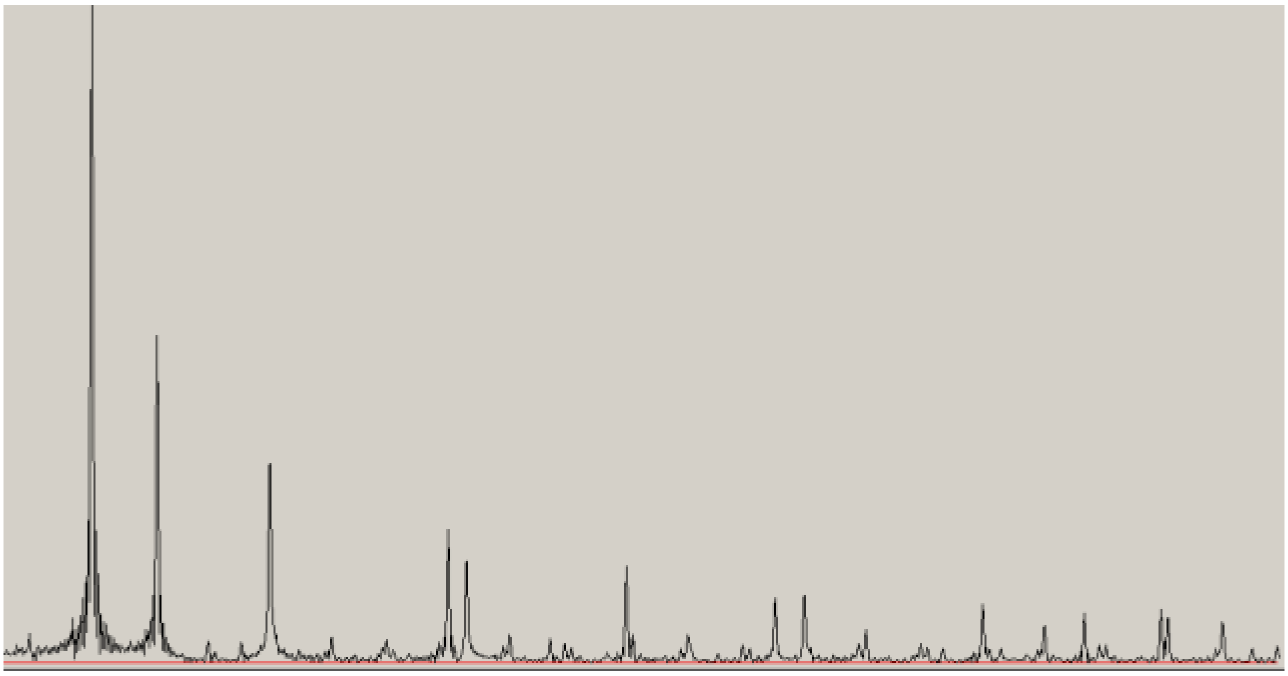


Fig. 14-3 Spectrum of the signal of Fig. 14-2

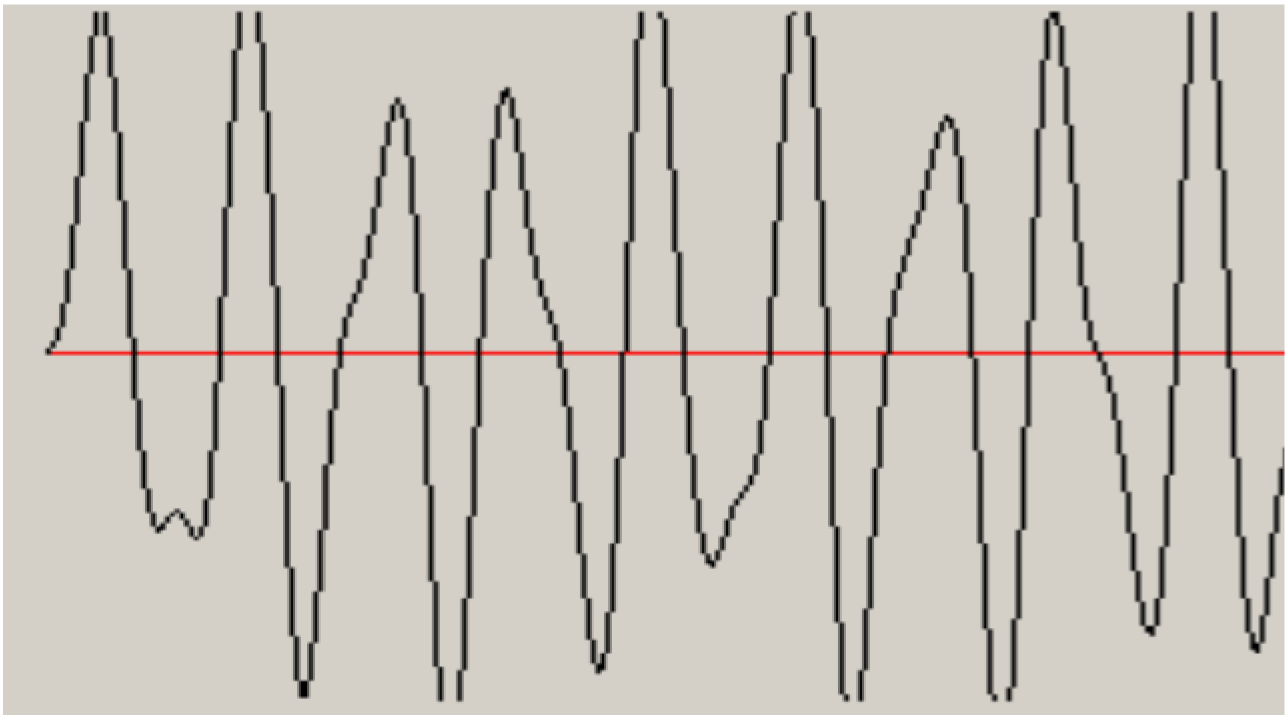


Fig. 14-4 Filtered signal

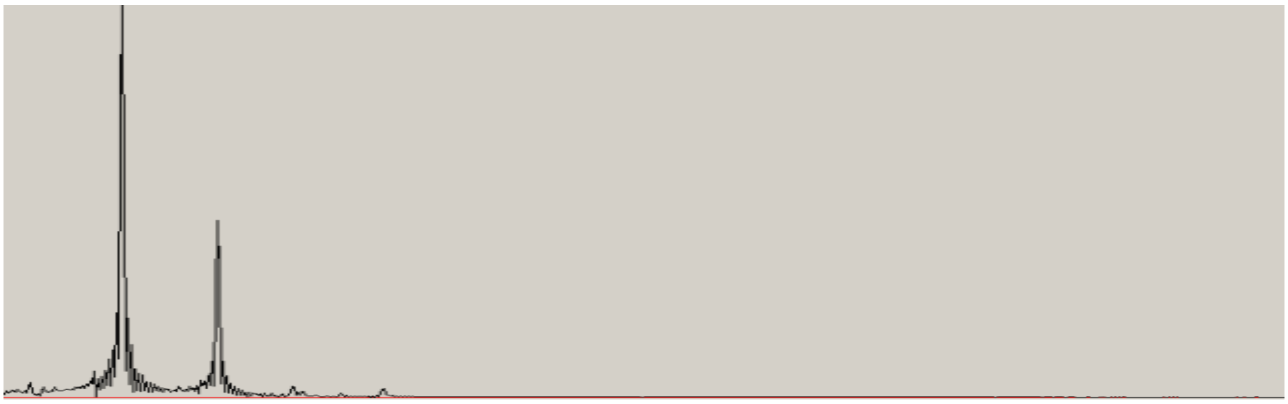


Fig. 14-5 Spectrum of the signal after filtering (output signal)

For generation of the filter coefficients use the Filter Designer Tool which is a constituent part of mikroC for dsPIC.

The program for generation of the described signal is as follows:

```
/* This project is designed to work with PIC P30F6014A. It has been tested


   on dsPICPRO3 board with 10.0 MHz crystal and 8xPLL. It should work with
any


   other crystal. Note: the maximum operating frequency for dsPIC is
120MHz.


   With minor adjustments, this example should work with any other dsPIC
MCU






   On-board DAC module


   Enable SPI connection to DAC on SW4 and DAC's Load(LD) and Chip
Select(CS) pins on SW3.


 */






#include <spi_const.h>                       // to be used with
spi_init_advanced routine






// *** Filter Designer Tool outputs *** //


const unsigned int BUFFER_SIZE  = 8;


const unsigned int FILTER_ORDER = 3;


const signed int   COEFF_B[FILTER_ORDER+1] = {0x21F3, 0x65DA, 0x65DA,
0x21F3};


const signed int   COEFF_A[FILTER_ORDER+1] = {0x2000, 0xB06D, 0x47EC,
0xE8B6};


const unsigned int SCALE_B =  6;


const unsigned int SCALE_A = -2;
```

```
// *** DAC pinout *** //

const char CS_PIN   = 1;                   // DAC CS pin

const char LOAD_PIN = 2;                   // DAC LOAD pin



unsigned int THalf_Low, THalf_High;        // half-periods of low and
high-frequency

                                           // square signals

char         char2send;                    // char recived from UART

unsigned int sample, sending_ch_cnt;       // digital signal sample,
sending char counter

unsigned int us_cntL, us_cntH;             // low and high-frequency
square signal

                                           // microseconds counters

signed int   input[BUFFER_SIZE];           // filter input signal (two
square signals)

signed int   output[BUFFER_SIZE];          // filtered signal

unsigned int sample_index;                 // index of current sample

signed int   voltageL, voltageH;           // square signals amplitudes



void InitMain() {



  LATC.CS_PIN    = 1;                      // set DAC CS to inactive

  LATC.LOAD_PIN  = 0;                      // set DAC LOAD to inactive

  TRISC.LOAD_PIN = 0;                      // configure DAC LOAD pin as
output
```

```
  TRISC.CS_PIN   = 0;                        // configure DAC CS pin as
output



  // Initialize SPI2 module

  Spi2_Init_Advanced(_SPI_MASTER, _SPI_16_BIT, _SPI_PRESCALE_SEC_1,
_SPI_PRESCALE_PRI_1,

                     _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE,
_SPI_CLK_IDLE_HIGH,

                     _SPI_ACTIVE_2_IDLE);




  Uart1_Init(9600);                          // Initialize UART1 module

}



void DAC_Output(unsigned int valueDAC) {



  LATC.CS_PIN = 0;                           // CS enable for DAC

  // filter output range is 16-bit number; DAC input range is 12-bit number

  valueDAC = valueDAC >> 4;

  // now both numbers are 12-bit, but filter output is signed and DAC input
is unsigned.

  //  Half of DAC range 4096/2=2048 is added to correct this

  valueDAC = valueDAC + 2048;

  SPI2BUF = 0x3000 | valueDAC;               // write valueDAC to DAC (0x3
is required by DAC)
```

```
  while (SPI2STAT.F1)                        // wait for SPI module to
finish sending

    asm nop;

  LATC.CS_PIN = 1;                           // CS disable for DAC

}



void SetPeriods(char ch) {



/* DTMF frequencies:



            1209 Hz     1336 Hz     1477 Hz     1633 Hz

 697 Hz       1           2           3           A

 770 Hz       4           5           6           B

 852 Hz       7           8           9           C

 941 Hz       *           0           #           D

*/



  // Calculate half-periods in microseconds

  //   example: 1/697Hz = 0.001435 seconds = 1435 microseconds

  //            1435/2  = 717



  switch (ch) {

    case 49: THalf_Low = 717; THalf_High = 414; break;  //'1'
    case 50: THalf_Low = 717; THalf_High = 374; break;  //'2'

    case 51: THalf_Low = 717; THalf_High = 339; break;  //'3'

    case 65: THalf_Low = 717; THalf_High = 306; break;  //'A'



    case 52: THalf_Low = 649; THalf_High = 414; break;  //'4'

    case 53: THalf_Low = 649; THalf_High = 374; break;  //'5'

    case 54: THalf_Low = 649; THalf_High = 339; break;  //'6'

    case 66: THalf_Low = 649; THalf_High = 306; break;  //'B'



    case 55: THalf_Low = 587; THalf_High = 414; break;  //'7'

    case 56: THalf_Low = 587; THalf_High = 374; break;  //'8'

    case 57: THalf_Low = 587; THalf_High = 339; break;  //'9'

    case 67: THalf_Low = 587; THalf_High = 306; break;  //'C'



    case 42: THalf_Low = 531; THalf_High = 414; break;  //'*'

    case 48: THalf_Low = 531; THalf_High = 374; break;  //'0'

    case 35: THalf_Low = 531; THalf_High = 339; break;  //'#'

    case 68: THalf_Low = 531; THalf_High = 306; break;  //'D'

  }

}



void ClearBufs() {
  //Clear buffers

  Vector_Set(input,  BUFFER_SIZE, 0);

  Vector_Set(output, BUFFER_SIZE, 0);

}



void Timer1Int() org 0x1A {            // interrupt frequency is 20kHz



  // calculate sample

  sample = voltageL + voltageH;        // add voltages

  input[sample_index] = sample;        // write sample to input buffer



  // update low-frequency square signal microseconds counter

  us_cntL = us_cntL + 50;              // since us_cntL and THalf_Low are
in microseconds

                                       // and Timer1 interrupt occures
every 50us

                                       // increment us_cntL by 50

  if (us_cntL > THalf_Low) {           // half-period exceeded, change sign

    voltageL = -voltageL;

    us_cntL  = us_cntL - THalf_Low;    // subtract half-period

  }



  // update high-frequency square signal microseconds counter

  us_cntH = us_cntH + 50;
  if (us_cntH > THalf_High) {

    voltageH = -voltageH;

    us_cntH  = us_cntH - THalf_High;

  }



  //IIR(amp), filtering new sample

  sample = IIR_Radix(SCALE_B, SCALE_A, COEFF_B, COEFF_A, FILTER_ORDER+1,

                   input, BUFFER_SIZE, output, sample_index);



  DAC_Output(sample);                 // send sample to digital-to-analog
converter



  output[sample_index] = sample;      // write filtered sample in output
buffer



  sample_index++;                     // increment sample index, prepare
for next sample

  if (sample_index >= BUFFER_SIZE)

    sample_index = 0;



  sending_ch_cnt--;                   // decrement char sending counter

                                      //   (character transmition lasts
90ms = 1800 samples)

  if (sending_ch_cnt == 0) {          // if character transmition is over

    T1CON=0;                          // turn off Timer1
    Delay_ms(200);                         // pause between two characters is
200ms

  }



  IFS0.F3 = 0;                             // clear Timer1 interrupt flag

}





// --- main --- //

void main() {



  InitMain();                              // perform initializations



  sending_ch_cnt = 0;                      // reset counter

  sample_index   = 0;                      // initialize sample index



  // Clear interrupt flags

  IFS0 = 0;

  IFS1 = 0;

  IFS2 = 0;



  INTCON1 = 0x8000;                        // disable nested interrupts

  IEC0    = 0x0008;                        // enable Timer1 interrupt
  // Timer1 input clock is Fosc/4. Sampling frequency is 20kHz. Timer
should

  // raise interrupt every 50 microseconds. PR1 = (Fosc[Hz]/4) / 20000Hz =
Fosc[kHz]/(4*20)

  PR1 = Clock_kHz() / 80;

  // Note: interrupt routine execution takes ~10us



  while (1) {                          // endless loop

    if ((sending_ch_cnt == 0) &&       // check if sending of previous
character is over

        (Uart1_Data_Ready() > 0)) {    // check if character arrived via
UART1



      char2send = Uart1_Read_Char();   // read data from UART and store it

      SetPeriods(char2send);           // set periods for low and high-
frequency square signals

      ClearBufs();                     // clear input and output buffers

      // digital filter computing error is smaller for signals of higher
amplitudes

      // so signal amplitude should as high as possible. The highest value
for

      // signed integer type is 0x7FFF but since we are adding 2 signals we
must

      // divide it by 2.

      voltageH = 0x7FFF / 2;           // high-frequency square signal
amplitude

      voltageL = 0x7FFF / 2;           // low-frequency square signal
amplitude
```

```
      us_cntL = 0;                    // low-frequency square signal
microseconds counter


      us_cntH = 0;                    // high-frequency square signal
microseconds counter




      // start Timer T1


      sending_ch_cnt =   1800;        // character tansmition lasts 90ms =
1800 samples * 50us


      T1CON         = 0x8000;         // enable Timer1 (TimerOn, prescaler
1:1)


    }


  }


}
```

14.2.3 Example 3 – DTMFin

The example shows a method of detecting DTMF signal.

The following detection algorithm has been applied.

The level of input signal denoting the presence of DTMF signal is awaited. Frequency estimation over 1024 samples is performed. The sampling frequency is 20kHz. In other words, the process of estimation lasts approximately 50ms. The minimum length of one character is 65ms. The minimum pause after one character is 80ms. For this reason each estimation is followed by an 80ms pause, and then the next character is awaited.

The estimate of the signal frequency can be performed by counting zero crossings, as shown in Fig. 14-6.

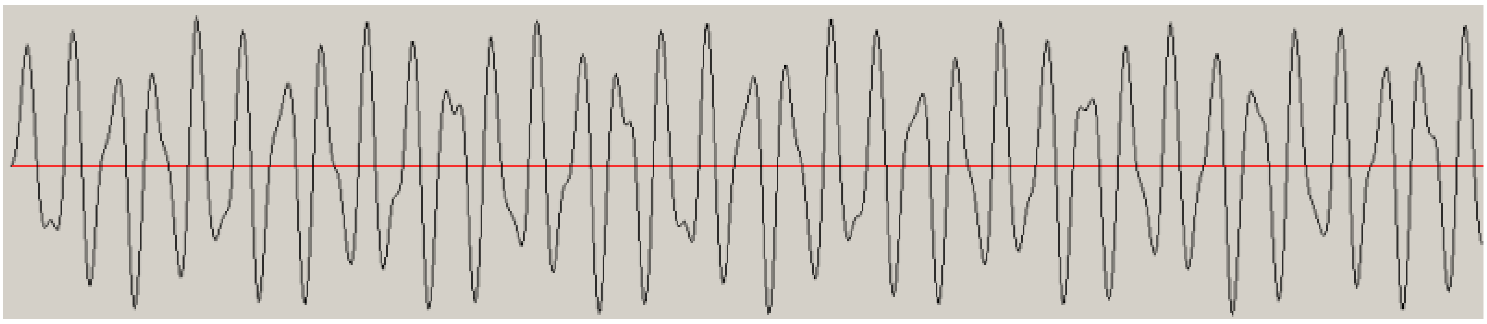


Fig. 14-6 DTMF signal before filtering

Before zero crossing estimation algorithm can be performed signal must be filtred to separate low frequency from high frequency.

The signal is then put through a lowpass IIR filter of the 4th order having stopband corner frequency 1200Hz whereby all frequencies except the lower frequency are suppressed.

By counting zero crossings of the filtered signal the estimation of the frequency of the lower frequency signal is performed.

The signal is also put through a highpass IIR filter of the 4th order having stopband corner frequency 950Hz whereby all frequencies except the higher frequency are suppressed. The signal after filtering is shown in Fig. 14-7.

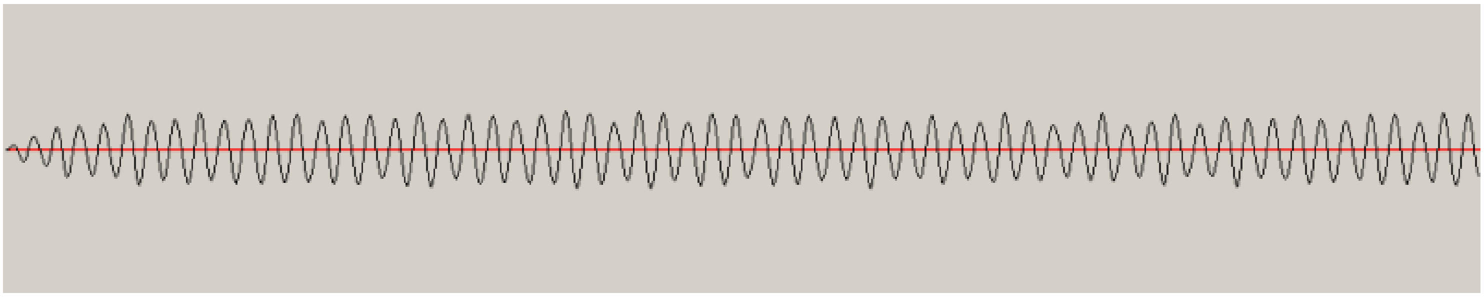
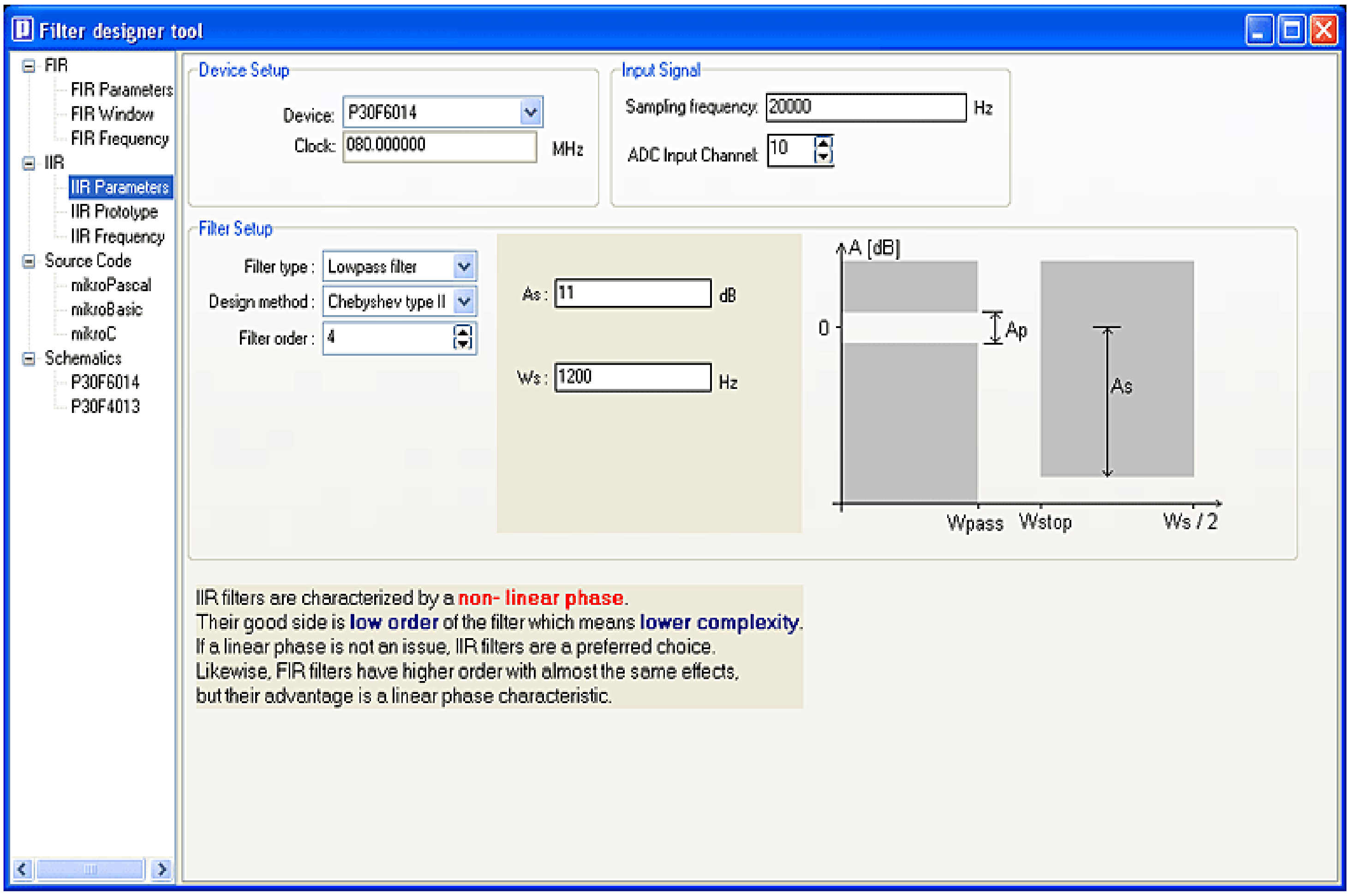


Fig. 14-7 DTMF signal after filtering

By counting zero crossings of the filtered signal the estimation of the frequency of the higher frequency signal is performed. After the frequencies have been obtaind, the sent character is obtained simply by comparison.

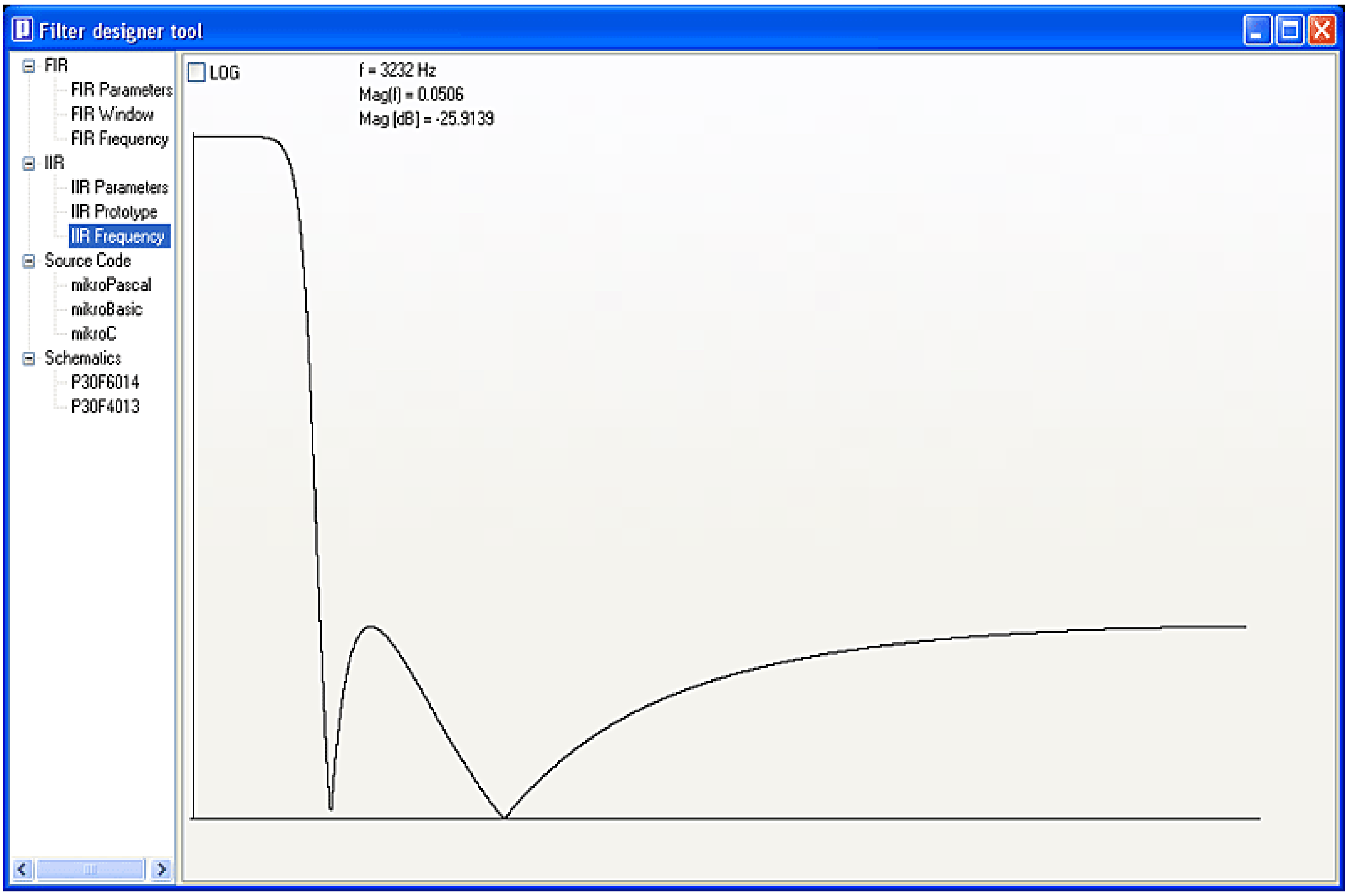
For generation of the filter coefficients one can use the Filter Designer Tool which is a constituent part of mikroC for dsPIC. Settings of Filter Designer Tool for LowPass and HighPass filter are given below.



Device: P30F6014
Clock: 080.000000 MHz
Sampling frequency: 20000 Hz
ADC Input Channel: 10
Filter type : Lowpass filter
Design method : Chebyshev type II
Filter order : 4
As : 11 dB
Ws : 1200 Hz

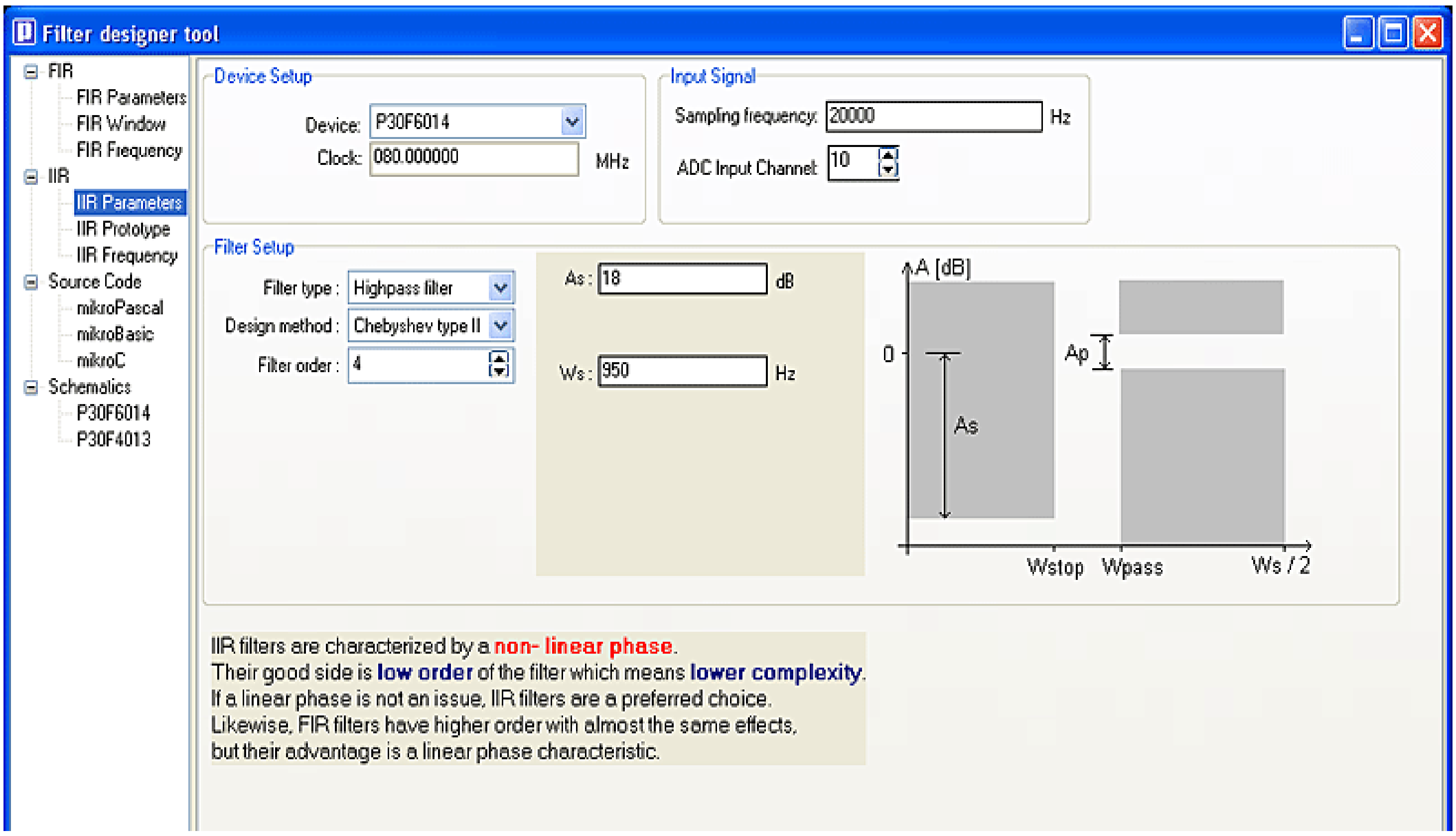
IIR filters are characterized by a **non- linear phase**.
Their good side is **low order** of the filter which means **lower complexity**.
If a linear phase is not an issue, IIR filters are a preferred choice.
Likewise, FIR filters have higher order with almost the same effects,
but their advantage is a linear phase characteristic.

IIR lowpass filter settings



f = 3232 Hz
Mag(f) = 0.0506
Mag [dB] = -25.9139

IIR lowpass frequency window



Device Setup

Device: P30F6014
Clock: 080.000000 MHz

Input Signal

Sampling frequency: 20000 Hz
ADC Input Channel: 10

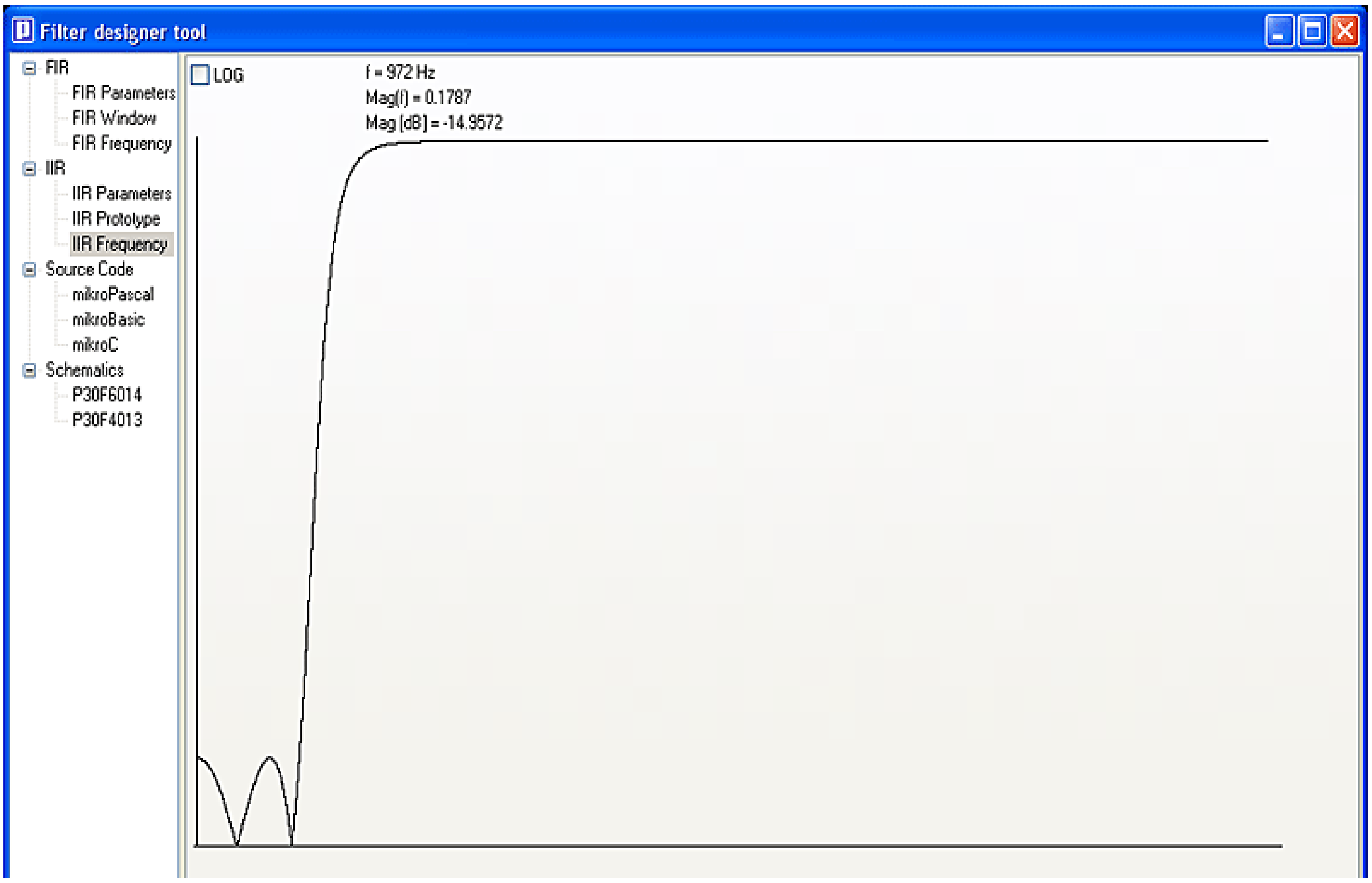
Filter Setup

Filter type : Highpass filter
Design method : Chebyshev type II
Filter order : 4

As : 18 dB
Ws : 950 Hz

IIR filters are characterized by a non- linear phase.
Their good side is low order of the filter which means lower complexity.
If a linear phase is not an issue, IIR filters are a preferred choice.
Likewise, FIR filters have higher order with almost the same effects,
but their advantage is a linear phase characteristic.

IIR highpass filter settings



IIR highpass frequency window

Implementation of the described algorithm:

```
/* This project is designed to work with PIC P30F6014A. It has been tested


   on dsPICPRO3 board with 10.0 MHz crystal and 8xPLL. It should work with
any


   other crystal. Note: the maximum operating frequency for dsPIC is
120MHz.


   With minor adjustments, this example should work with any other dsPIC
MCU


*/




#include <Spi_Const.h>




// *** DAC pinout *** //
```

```
const  LOAD_PIN = 2;                         // DAC load pin

const  CS_PIN   = 1;                         // DAC CS pin






// filter setup:

//     filter kind: IIR

//     filter type: lowpass filter

//     filter order: 4

//     design method: Chebyshev type II

const unsigned int  BUFFER_SIZE   = 8;

const unsigned int  FILTER_ORDER  = 4;

const unsigned int  BPF1_COEFF_B[FILTER_ORDER+1] = {0x1BD7, 0xAB5D, 0x753A,
0xAB5D, 0x1BD7};

const unsigned int  BPF1_COEFF_A[FILTER_ORDER+1] = {0x2000, 0xA1C7, 0x6C59,
0xC6EA, 0x0BDE};

const unsigned int  BPF1_SCALE_B       =  0;

const unsigned int  BPF1_SCALE_A       = -2;

// filter setup:

//     filter kind: IIR

//     filter type: Highpass filter

//     filter order: 4

//     design method: Chebyshev type II
```

```
const unsigned int  BPF2_COEFF_B[FILTER_ORDER+1] = {0x0BF7, 0xD133, 0x45AF,
0xD133, 0x0BF7};


const unsigned int  BPF2_COEFF_A[FILTER_ORDER+1] = {0x1000, 0xCA8B, 0x44B5,
0xD7E5, 0x08F3};


const unsigned int  BPF2_SCALE_B       = -3;


const unsigned int  BPF2_SCALE_A       = -3;





// min voltage offset level on ADC that can be detected as DTMF


const unsigned int  MinLevel = 18;







char SignalActive;                    // indicator (if input signal exists)


int  sample;                          // temp variable used for reading
from ADC


char  Key;                            // detected character


long int  f;                          // detected frequency


unsigned  SampleCounter;              // indicates the number of samples in
circular buffer


unsigned  sample_index;               // index of next sample


int  input[8];                        // circular buffer - raw samples
(directly after ADC)


int  output_f1[8];                    // circular buffer - samples after
IIR BP filter


int  output_f2[8];                    // circular buffer - samples after
IIR BP filter




unsigned  TransitLow, TransitHigh;    // counts of transitions (low, high
freq)
```

```
int  sgnLow, sgnHigh;                  // current signs of low and high freq
signal

int  KeyCnt;                           // number of recived DTFM and
displayed on LCD




void Estimate(){

unsigned fd;

/* DTMF frequencies:

                1209 Hz     1336 Hz     1477 Hz     1633 Hz

      697 Hz      1           2           3           A

      770 Hz      4           5           6           B

      852 Hz      7           8           9           C

      941 Hz      *           0           #           D

*/



  // calculating index of lower freq

  f = TransitLow*20000l;  // f = No_Of_Transitions*Sampling_Freq [Hz]

  f = f >> 11;            // f = f div 2048 = f/2/1024 (2 transitions in
each period)



  if (f < 733)

    fd = 1;   //Index of Low_freq = 1

  else if (f < 811)
```

```
    fd = 2;   //Index of Low_freq = 2

  else if (f < 896)

    fd = 3;   //Index of Low_freq = 3

  else

    fd = 4;  //Index of Low_freq = 4



  // calculating index of higher freq

  f = TransitHigh*20000l;  // f = No_Of_Transitions*Sampling_Freq

  f = f >> 11;             // f = f/2048 = f/2/1024 (2 transitions in each
period)



  if (f<1272)

    fd = fd + 10;  // encode Index of higher freq as 10

  else if (f<1406)

    fd = fd + 20;  // encode Index of higher freq as 20

  else if (f<1555)

    fd = fd + 30;  // encode Index of higher freq as 30

  else

    fd = fd + 40; // encode Index of higher freq as 40



  switch (fd){    // reading of input char from DTMF matrix

   case 11: Key = '1'; break;

   case 12: Key = '4'; break;
```

```
     case 13: Key = '7'; break;

     case 14: Key = '*'; break;

     case 21: Key = '2'; break;

     case 22: Key = '5'; break;

     case 23: Key = '8'; break;

     case 24: Key = '0'; break;

     case 31: Key = '3'; break;

     case 32: Key = '6'; break;

     case 33: Key = '9'; break;

     case 34: Key = '#'; break;

     case 41: Key = 'A'; break;

     case 42: Key = 'B'; break;

     case 43: Key = 'C'; break;

     case 44: Key = 'D'; break;

     }



    // diplay recived char on second row of LCD

    if(KeyCnt >= 16)

      { // if second row is full erase it and postion cursor at first column

        Lcd_Custom_Cmd(LCD_SECOND_ROW);

        Lcd_Custom_Out_CP("                ");

        Lcd_Custom_Cmd(LCD_SECOND_ROW);
        KeyCnt = 0; // reset recived DTFM signals counter

      }

    Lcd_Custom_Chr_CP(Key); // output recived on LCD

    KeyCnt++;               // increment counter



  }



  void DAC_Output(unsigned valueDAC){

    LATC.CS_PIN = 0;  // CS enable for DAC

    // filter output range is 16-bit number; DAC input range is 12-bit number

    valueDAC = valueDAC >> 4;

    // now both numbers are 12-bit but filter output is signed and DAC input
  is unsigned.

    // half of DAC range 4096/2=2048 is added to correct this

    valueDAC = valueDAC + 2048;

    SPI2BUF = 0x3000 | valueDAC;         // write valueDAC to DAC (0x3 is
  required by DAC)

    while (SPI2STAT.f1)                  // wait for SPI module to finish
  sending

      asm nop;

    LATC.CS_PIN = 1;                     // CS disable for DAC

  }



  void InitDec(){
```

```
                                     // estimate on 1024 samples for fast
DIV

  SampleCounter = 1024;              // init low-freq transitions counter

  TransitLow = 0;                    // init high-freq transitions counter

  TransitHigh = 0;                   // init input circular buffer (zero-
filled)

  Vector_Set(input, 8, 0);           // init filtered circular buffer
(zero-filled)

  Vector_Set(output_f1, 8, 0);       // init filtered circular buffer
(zero-filled)

  Vector_Set(output_f2, 8, 0);       // points on first element of circular
buffer

  sample_index = 0;                  // current sign is positive

  sgnLow = 0;                        // current sign is positive

  sgnHigh = 0;

  DAC_Output(0);

}



void ADC1Int() org 0x2A {

  sample = ADCBUF0;                                 // read input ADC signal



  if ((sample > 2048+MinLevel) &&  !SignalActive)  // detecting signal

    {

      SignalActive = 1;                             // activate estimation
algorithm

      InitDec();                                    // initialize variables
    }



  // since ADC is configured to get samples as intgers

  //   mean value of input signal is expected to be located at

  //   middle of ADC voltage range

  sample = sample << 4;

  sample = sample-(2048 << 4);  // expanding signal to full scale

  // now sample is ready to be filtred





  if (SignalActive)

    {

      input[sample_index] = sample;  // write sample in circular buffer



      // filter input signal (for low-freq estimation)

      sample  = IIR_Radix(BPF1_SCALE_B, BPF1_SCALE_A, BPF1_COEFF_B,
BPF1_COEFF_A,

                          FILTER_ORDER+1, input, BUFFER_SIZE, output_f1,
sample_index);

      DAC_Output(sample);  // output filtred signal to DAC for Visual check

      output_f1[sample_index] = sample;



      // transition_Low?

      if ((sample & 0x8000) != sgnLow)     // if transition trough 0
```

```
        {

          sgnLow = (sample & 0x8000);      // save current sign

          ++TransitLow;                    // increment transition counter

        }



      // filter input signal (for high-freq estimation)

      sample  = IIR_Radix(BPF2_SCALE_B, BPF2_SCALE_A, BPF2_COEFF_B,
BPF2_COEFF_A,

                          FILTER_ORDER+1, input, BUFFER_SIZE, output_f2,
sample_index);



      output_f2[sample_index] = sample;      // write filtered signal in
buffer

      // transition_High?

      if ((sample & 0x8000) != sgnHigh)      // if transition

        {

          sgnHigh = (sample & 0x8000);

          ++TransitHigh;                     // increment transition
counter

        }



      sample_index = (sample_index+1) & 7;   // move pointer on next
element

      --SampleCounter;                       // decrement sample counter

      if (SampleCounter == 0)                // if all of 1024 samples are
readed
```

```
        {

          SignalActive = 0;                  // deactivate estimation
algorithm

          Estimate();                        // read estimated character

          DAC_Output(0);                     // set DAC output to 0

          Delay_ms(80);                      // wait for next char

        }

    }

  IFS0.f11 = 0;                              // clear ADC complete IF

}



void Timer1Int() org 0x1A{



  ADCON1.f1  = 1; // ASAM=0 and SAMP=1 begin sampling

  ADCON1.f15 = 1; // start ADC

  IFS0.f3    = 0; // clear Timer1 IF

}



void main(){

  KeyCnt = 0;                                              // set to 0

  SignalActive = 0;                                        // no signal is
present

  ADPCFG = 0xFFFF;                                         // configure
pins as digital
```

```
  Lcd_Custom_Config(&PORTD, 7, 6, 5, 4, &PORTB, 4, 2, 6);   // initialize
LCD

  Lcd_Custom_Out(1,1,"tone is:");                           // print
message at first row

  Lcd_Custom_Cmd(Lcd_SECOND_ROW);                           // position
cursor at second row

  LATC.CS_PIN    = 1;                                       // set DAC CS
to inactive

  LATC.LOAD_PIN  = 0;                                       // set DAC LOAD
to inactive

  TRISC.LOAD_PIN = 0;                                       // configure
DAC LOAD pin as output

  TRISC.CS_PIN   = 0;                                       // configure
DAC CS pin as output

  // Initialize SPI2 module

  Spi2_Init_Advanced(_SPI_MASTER, _SPI_16_BIT, _SPI_PRESCALE_SEC_1,
_SPI_PRESCALE_PRI_1,

                     _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE,
_SPI_CLK_IDLE_HIGH,

                     _SPI_ACTIVE_2_IDLE);



  TRISB.f10 = 1;           // configure RB10 pin as input

  ADPCFG    = 0xFBFF;      // configure RB10 pin as analog

  ADCON1    = 0x00E0;      // auto-convert, auto-conversion

  ADCON2    = 0x0000;

  ADCON3    = 0x021A;      // sampling time=2*Tad, minimum Tad selected

  ADCHS     = 0x000A;      // sample input on RB10

  ADCSSL    = 0;           // no input scan
  // clear interrupt flags

  IFS0    = 0;

  IFS1    = 0;

  IFS2    = 0;



  INTCON1 = 0x8000;      // disable nested interrupts

  INTCON2 = 0;

  IEC0    = 0x0808;      // enable Timer1 and ADC interrupts

  IPC0.f12 = 1;          // Timer1 interrupt priority level = 1

  IPC2.f13 = 1;          // ADC interrupt priority level = 2



  // timer1 input clock is Fosc/4. Sampling frequency is 20kHz. Timer
should

  // raise interrupt every 50 microseconds. PR1 = (Fosc[Hz]/4) / 20000Hz =
Fosc[kHz]/(4*20)

  PR1   = Clock_kHz() / 80;

  T1CON = 0x8000;    // Enable Timer1



  while(1)           // Infinite loop

    asm nop;

}
```

14.2.4 Example 4 – Acceleration sensor

This example shows a possibility of using dsPIC30F4013 or dsPIC6014A in order to realize control of the pointer of a GLCD on the basis of the signal from an acceleration sensor. In this example the use is made of an accelerometer card containing the sensor ADXL330 by Analog Devices. Besides the sensor, the card contains an operational amplifier used as a unit amplifier increasing the output current capacity of the sensor. The sensor measures the acceleration along two axes, X and Y. The offset voltage of the sensor has to be measured during the calibration function. From the accelerometer card, two analogue signals, for the X and Y axes, are fed to the inputs of AN8 and AN9 AD converters (pins PORTB.8 and PORTB.9). After the sampling and conversion, the measured value is presented as the shift of the pointer from the central position on the GLCD.

This example is widely applicable for the realization of e.g. joysticks, simple gyroscopes, robot controls or movement detectors.

```
/*  An example of the use of the microcontroller dsPIC30F6014A and Accel
Extra Board.


    The example shows how the signal from the sensor is sampled and how the
information on the


    accelerations along the X and Y axes are used for controlling the
cursor on a GLCD.


    The example also covers the calibration of the sensor (determination of
zeroG


    and 1G values for X and Y axes). Pin RC1 is used as user input. Pull-
down PORTC and


    put button jumper in Vcc position. */







// --- GLCD Messages ---


const char msg1[] = "Put board to pos  ";


const char msg2[] = "and press RC1";




// Global variables


signed int zeroG_x, zeroG_y; // zero gravity values
signed int oneG_x,  oneG_y;  // 1G values

signed int meas_x,  meas_y;  // measured values

signed int box_x,   box_y;   // variables for drawing box on GLCD

char positionNo;             // variable used in text messages

char text[21];               // variable used for text messages



void Init() {

  ADPCFG   = 0xFCFF;         // configure AN8(RB8) and AN9(RB9) as analog
pins

  TRISB.F8 = 1;              // configure RB8 and RB9 as input pins

  TRISB.F9 = 1;

  Glcd_Init_DsPicPro3();     // init GLCD for dsPICPRO3 board

  // Note: GLCD/LCD Setup routines are in the setup library files located
in the Uses folder

  // These routines will be moved into AutoComplete in the future.

  Glcd_Set_Font(FontSystem5x8, 5, 8, 32); // set GLCD font

  Glcd_Fill(0);              // clear GLCD

  TRISC = 0x02;              // pin PORTC.1 is input for calibration

  positionNo = 1;            // variable used in text messages

}



void DoMeasureXY() {

  meas_x = Adc_Read(8);      // measure X axis acceleration

  meas_y = Adc_Read(9);      // measure Y axis acceleration
```

```
}



void DrawPointerBox() {

  float x_real, y_real;



  x_real = (float)(meas_x-zeroG_x)/(oneG_x-zeroG_x); // scale [-1G..1G] to
[-1..1]

  x_real = x_real * 64;                              // scale [-1..1]   to
[-64..64]

  x_real = x_real + 64;                              // scale [-64..64] to
[0..128]



  y_real = (float)(meas_y-zeroG_y)/(oneG_y-zeroG_y); // scale [-1G..1G] to
[-1..1]

  y_real = y_real * 32;                              // scale [-1..1]   to
[-32..32]

  y_real = y_real + 32;                              // scale [-32..32] to
[0..64]



  // convert reals to integers

  box_x = (int)x_real;

  box_y = (int)y_real;



  // force x and y to range [0..124] and [0..60] because of Glcd_Box
parameters range

  if (box_x>124) box_x=124;

  if (box_x<0)   box_x=0;
  if (box_y>60)  box_y=60;

  if (box_y<0)   box_y=0;



  Glcd_Box(box_x, box_y, box_x+3, box_y+3, 2);  // draw box pointer,
color=2(invert ecah dot)

}



void ErasePointerBox() {

  Glcd_Box(box_x, box_y, box_x+3, box_y+3, 2);  // draw inverted box at the
same position

                                                // (erase box)

}



// --- Calibration procedure determines zeroG and 1G values for X and Y
axes ---//

void DoCalibrate() {

//  1) Put the Accel board in the position 1 : PARALLEL TO EARTH'S SURFACE

//  to measure Zero Gravity values for X and Y

  strcpy(text, msg1);

  text[17] = positionNo + 48;

  Glcd_Write_Text(text,5,1,1);

  positionNo++;

  strcpy(text, msg2);

  Glcd_Write_Text(text,5,20,1);
  while (PORTC.F1 == 0) ;          // wait for user to press RC1 button

  DoMeasureXY();

  zeroG_x = meas_x;                // save Zero Gravity values

  zeroG_y = meas_y;

  Delay_ms(1000);



//  2) Put the Accel board in the position 2 : X AXIS IS VERTICAL, WITH X
LABEL UP

//  to measure the 1G X value

  strcpy(text, msg1);

  text[17] = positionNo + 48;

  Glcd_Write_Text(text,5,1,1);

  positionNo++;

  strcpy(text, msg2);

  Glcd_Write_Text(text,5,20,1);

  while (PORTC.F1 == 0) ;          // wait for user to press RC1 button

  DoMeasureXY();

  oneG_x = meas_x;                 // save X axis 1G value

  Delay_ms(1000);



//  3) Put the Accel board in the position 3 : Y AXIS IS VERTICAL, WITH Y
LABEL UP

//  to measure the 1G Y value

  strcpy(text, msg1);
  text[17] = positionNo + 48;

  Glcd_Write_Text(text,5,1,1);

  positionNo++;

  strcpy(text, msg2);

  Glcd_Write_Text(text,5,20,1);

  while (PORTC.F1 == 0) ;          // wait for user to press RC1 button

  DoMeasureXY();

  oneG_y = meas_y;                 // save Y axis 1G value

  Delay_ms(1000);

}




void main() {

  Init();                          // initialization



  DoCalibrate();                   // calibration



  Glcd_Fill(0);                    // clear GLCD

  Glcd_H_Line(0, 127, 32, 1);      // draw X and Y axes

  Glcd_V_Line(0, 63,  64, 1);

  Glcd_Write_Char('X', 122, 3, 1);

  Glcd_Write_Char('Y', 66,  0, 1);
  while (1) {                      // endless loop

    DoMeasureXY();                 // measure X and Y values

    DrawPointerBox();              // draw box on GLCD

    Delay_ms(250);                 // pause

    ErasePointerBox();             // erase box

  }

}
```

Chapter15: Digital filter design

This chapter briefly describes Filter Designer Tool which is a constituent part of the mikroPascal, mikroC, and mikroBasic. This tool serves for designing digital filters.

15.1 General concepts

Filter Designer Tool allows simple and very fast design of digital filters. Fig. 15-1 shows one option in the main menu of mikroC which enables access to Filter Designer Tool.

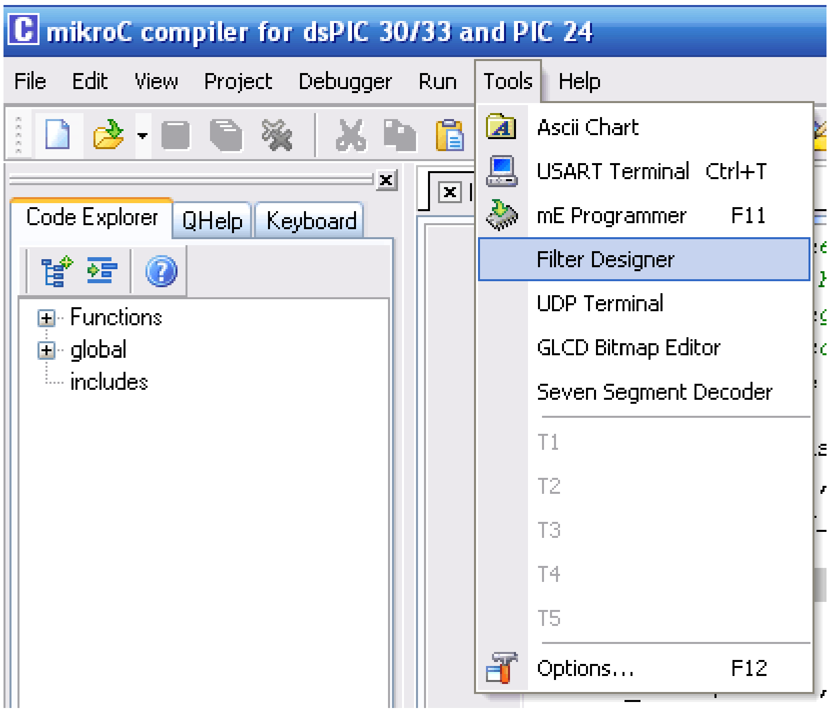


Fig. 15-1. Filter designer tool

There are two classes of digital filters. These are Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters. Both classes have their own merits and shortcomings. When designing a digital filter, the first task is to select the class of filter.

Table 15-1 presents the merits and shortcomings of FIR filters and, for comparison, Table 15-2 presents the merits and shortcomings of IIR filters.

Merits	**Shortcomings**
Stability condition is always fulfilled!	For a given amplitude characteristic FIR filter is of considerably higher order compared to IIR filter (higher complexity)
Linear phase (by a proper design applied in the Filter Designer Tool)	

Table 15-1. Merits and shortcomings of FIR filters

Merits	**Shortcomings**
For a given amplitude characteristic IIR filter is of considerably lower order	Stability condition is not always fulfilled. As the order of IIR filter increases, the probability that the

compared to FIR filter	filter will become unstable increases
	Nonlinear phase

Table 15-2. Merits and shortcomings of IIR filters

As one can see from the above Tables 15-1 and 15-2, a merit of one class of filters is a shortcoming of the other class. When starting a design, one should start from the type of signal to be filtered.

E.g. when processing audio signals, the phase is irrelevant. Therefore, the phase linearity can be sacrificed. By selecting the class of IIR filters, one can obtain a filter of lower order or a filter of the same order but of much higher selectivity compared to the corresponding class FIR filter.

For some other signals it may be necessary to preserve phase linearity. An example of such signals is the ECG signal. In this case IIR filter must not be used. However, the price for linearity is paid by a much higher order of the filter.

As the order of the filter increases, the corresponding signal processing will last longer and more memory will be required for the processing. For this reason it is necessary to select carefully the order of the filter and its input parameters. Fig. 15-2 shows the specifications for the amplitude characteristic of a digital filter. The class of filter (FIR or IIR) and design method will determine which input parameters will be available. The remaining parameters will be calculated by Filter Designer Tool to obtain the filter of the specified order.

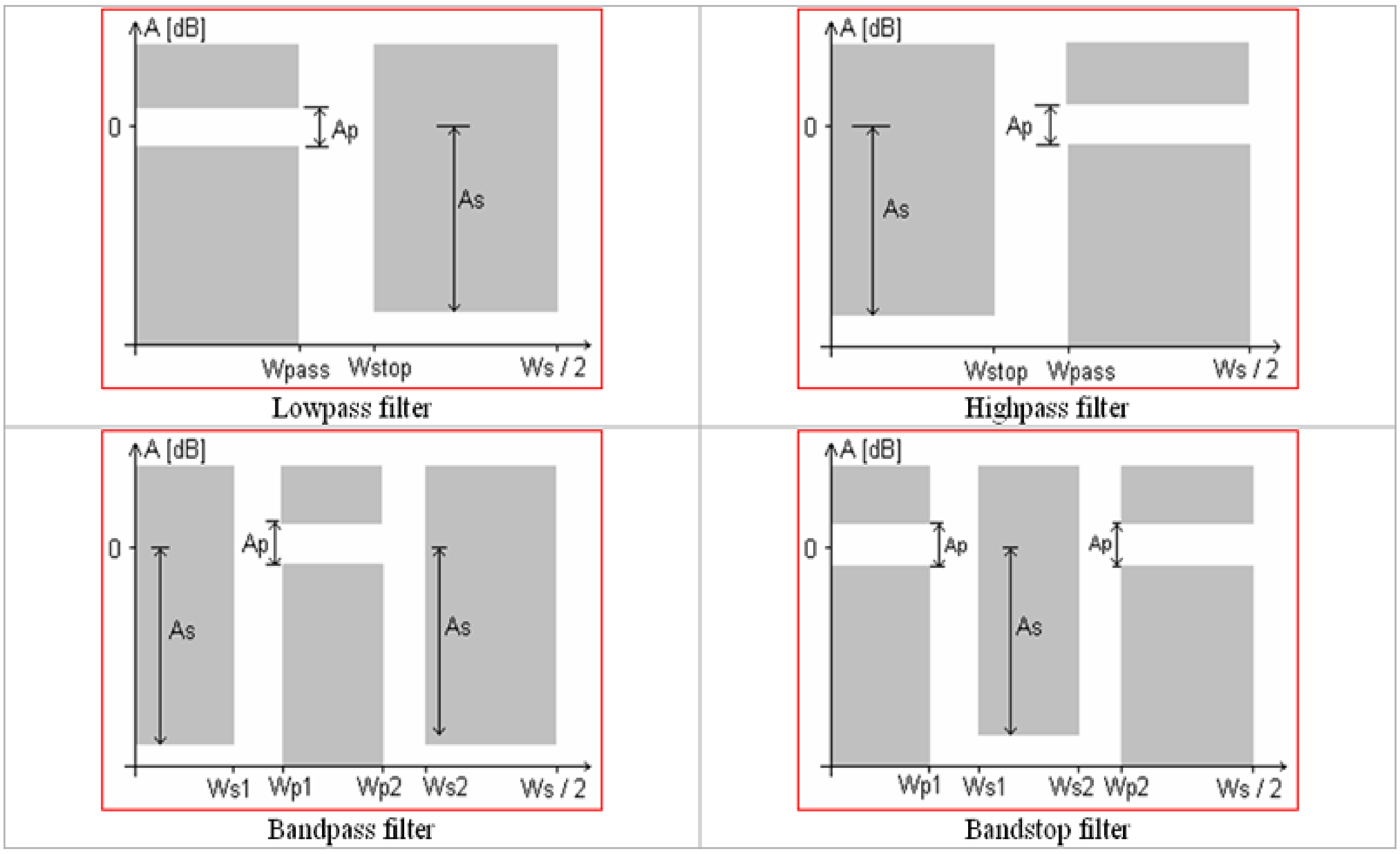


Fig. 15-2. Digital filter specifications

A[dB] – filter amplification in dB,
Ap – permitted bandpass ripple of amplification in dB,
As – minimum bandstop attenuation in dB,
Wp, Wpass – bandpass boundary frequencies,
Ws, Wpass – bandstop boundary frequencies,
Ws – one half of the sampling frequency.

- The frequency range 0 to Wpass is called passband. In the passband the maximum attenuation or amplification of signals is Ap.
- The frequency range Wpass to Wstop is called transition zone. In the transition zone the characteristic of the filter is unknown. It is only known that it is monotonically decreasing or monotonically increasing.
- The frequency range Wstop to Ws/2 is called bandstop. In the bandstop the signal is attenuated at least by As dB.

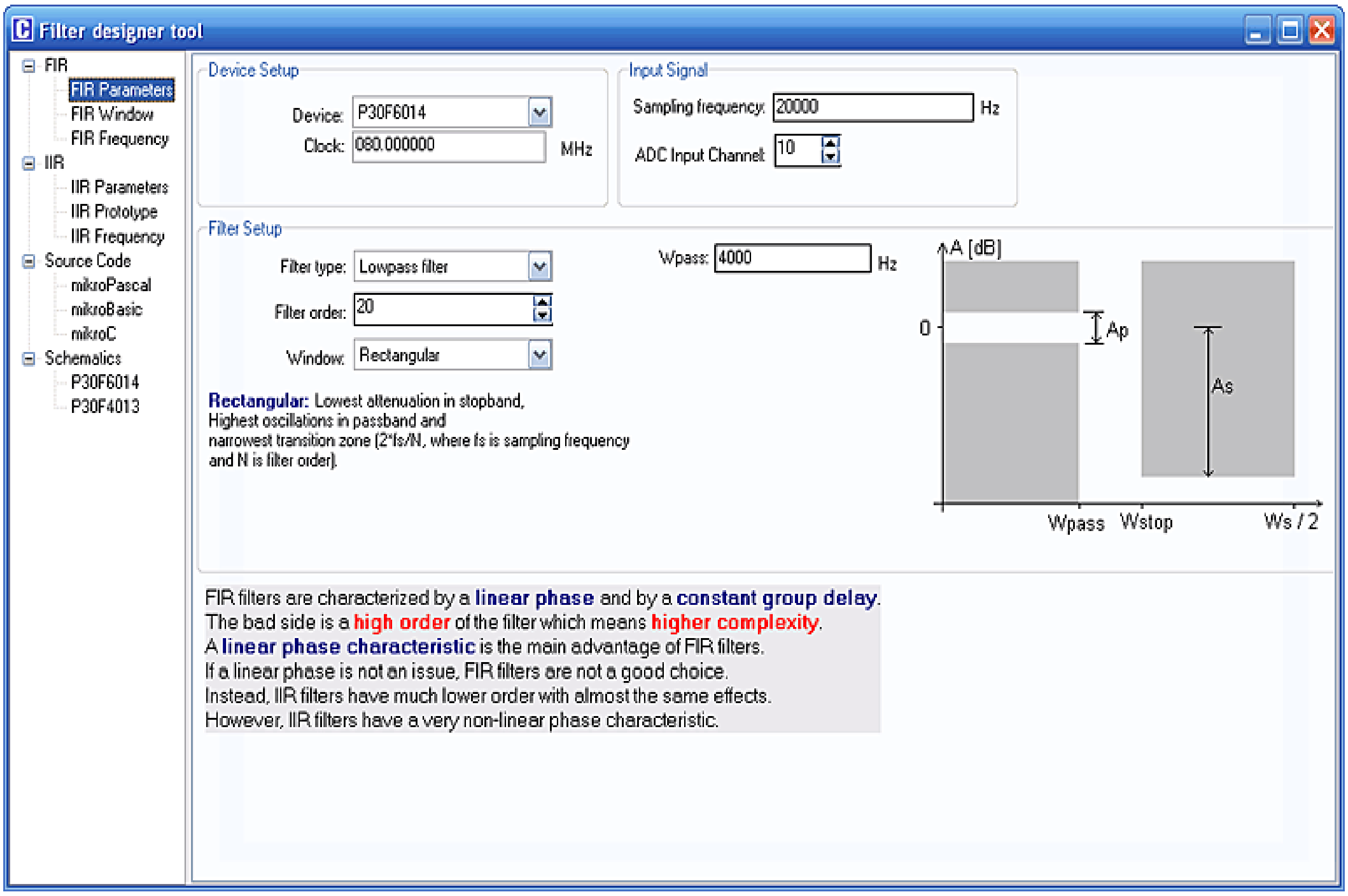
After the class (FIR or IIR) and input parameters are selected, designing of the desired filter is performed. In Chapter 15.2 the design of the finite impulse response (FIR) filters is presented. In Chapter 15.3 the design of the infinite impulse response (IIR) filters is presented.

15.2 FIR filters

FIR is the abbreviated notation for the finite impulse response digital filters. The basic properties of FIR filters are:

- relatively high order,
- linearity of phase,
- stability.

Design of a FIR filter starts by selecting Filter Designer Tool from the main menu of the mikroC compiler for dsPIC. The option FIR parameters on the left-hand side of the menu is selected. Then, the form shown in Fig. 15-3 appears.



Device: P30F6014
Clock: 080.000000 MHz

Sampling frequency: 20000 Hz
ADC Input Channel: 10

Filter type: Lowpass filter
Filter order: 20
Window: Rectangular
Wpass: 4000 Hz

Rectangular: Lowest attenuation in stopband,
Highest oscillations in passband and
narrowest transition zone (2*fs/N, where fs is sampling frequency
and N is filter order).

FIR filters are characterized by a **linear phase** and by a **constant group delay**.
The bad side is a **high order** of the filter which means **higher complexity**.
A **linear phase characteristic** is the main advantage of FIR filters.
If a linear phase is not an issue, FIR filters are not a good choice.
Instead, IIR filters have much lower order with almost the same effects.
However, IIR filters have a very non-linear phase characteristic.

Fig. 15-3 Entering FIR filter parameters

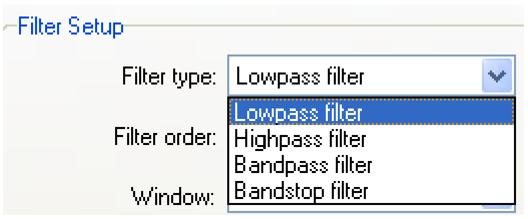
In the part **Device setup** appear the clock and designation of the used microcontroller.

In the part **Input Signal**, two parameters, sampling frequency and input channel of the AD converter, should be selected. The sampling frequency has to be at least twice as high as the maximum frequency of the input signal. If, e.g. the input signal is in the range 100Hz – 4400Hz, the minimum sampling frequency is 8800Hz. Of course, one never selects the minimum frequency, thus in this case the sampling frequency of at least 10kHz should be selected.

In the part Filter settings the type of filter is selected first:

- Lowpass filter,
- Highpass filter,
- Bandpass filter,
- Bandstop filter.

Then, the order of the filter is selected. This does not apply if Kaiser window function is applied, when on the basis of the selected parameters the order of filter is selected automatically. As the order of the filter increases, the selectivity of the filter will be higher (narrower transition zone), but the complexity of the filter will increase as a consequence of the more memory required for storing the samples and longer processing. The maximum sampling frequency depends on the length of processing (filter order) and selected clock.


Filter Setup

Filter type: Lowpass filter

Filter order:

Window:

Fig. 15-4 Type of FIR filter

After the type and order of the FIR filter are selected, the window function to be used can be selected. The bandstop attenuation and selectivity of the filter will depend on the window function. These two characteristics are mutually in conflict. A higher bandstop attenuation corresponds to a lower selectivity and vice versa. Kaiser window function is an optimum window function which gives the maximum attenuation for a given selectivity.

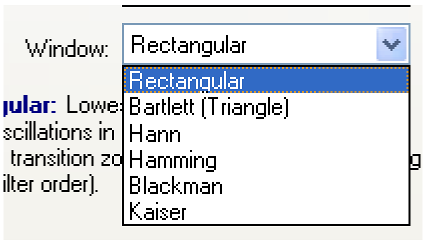

Window: Rectangular

Fig. 15-5 Selecting window function

There remains only one parameter to be selected and that is the boundary frequency of the filter.

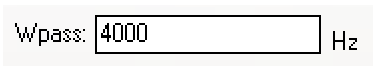

Wpass: 4000 Hz

Fig. 15-6 Boundary frequency of the FIR filter

When all FIR filter parameters have been entered, design of the filter is carried out simply by selecting one of the options on the left-hand side of Filter Designer Tool (FIR Window, FIR Frequency, mikroPascal, mikroBasic, mikroC).

In the part FIR Window the shape of the window function and pulse response of the designed filter can be observed. If a lowpass filter of the order 20 is selected, Hamming window function having the boundary frequency 4000Hz will result in the pulse response shown in Fig.15-7.

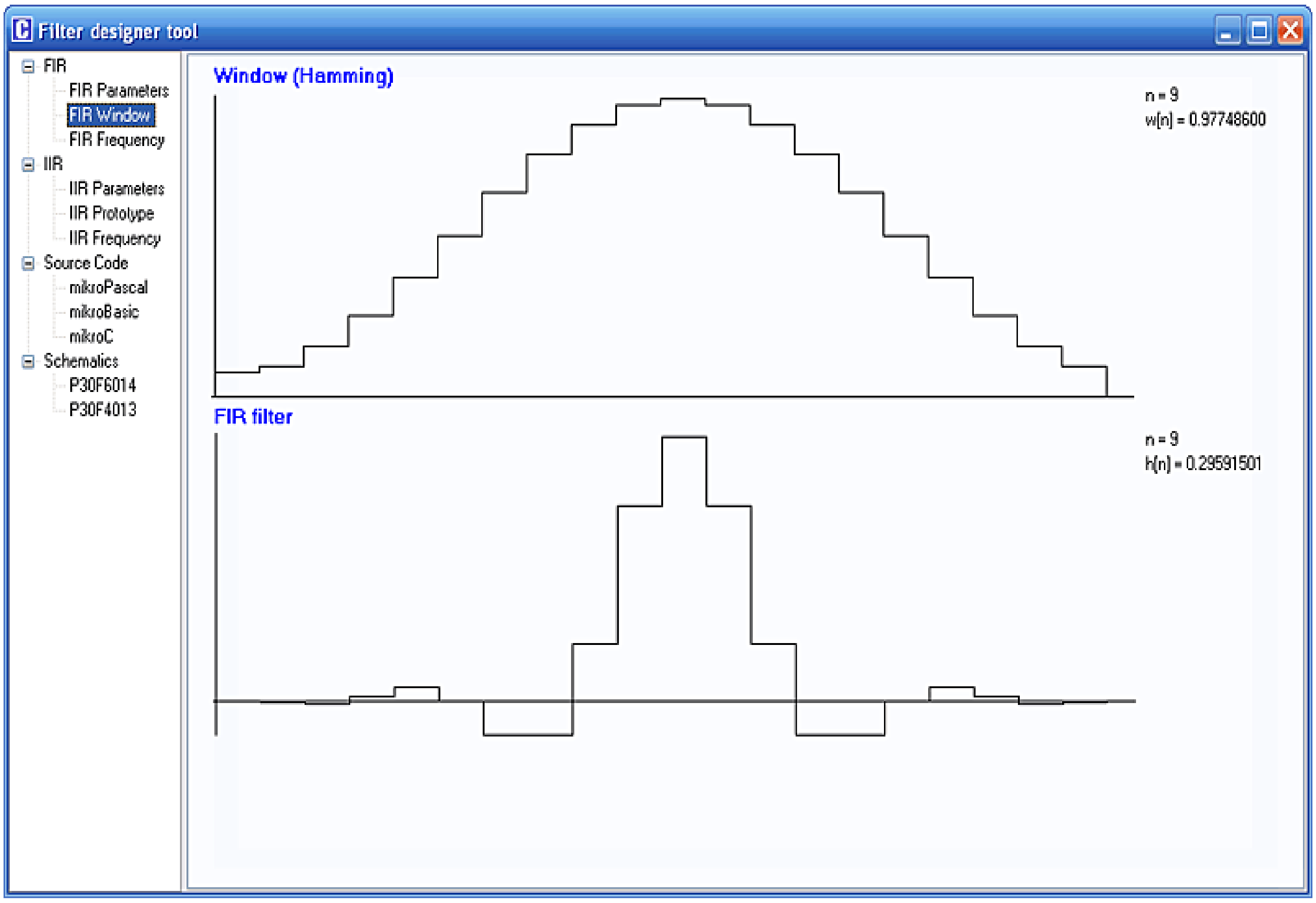


Fig. 15-7 Window function and pulse response of the designed filter

In the part FIR Frequency one can see the shape of the frequency characteristic of the window function and the frequency characteristic of the designed digital filter. If a lowpass filter of the order 20 is selected, Hamming window function having the boundary frequency 4000Hz will result in the pulse response shown in Fig.15-8.

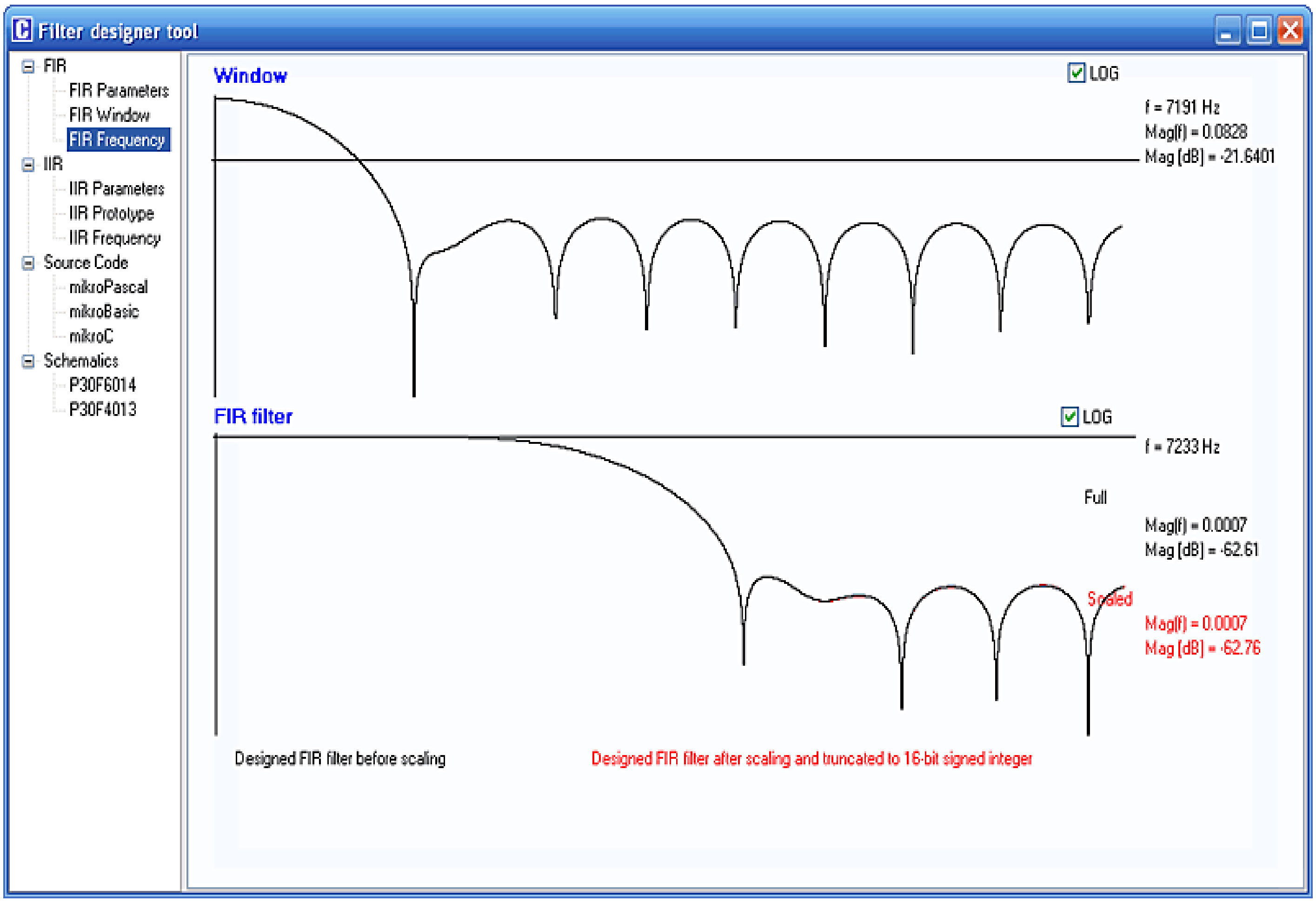
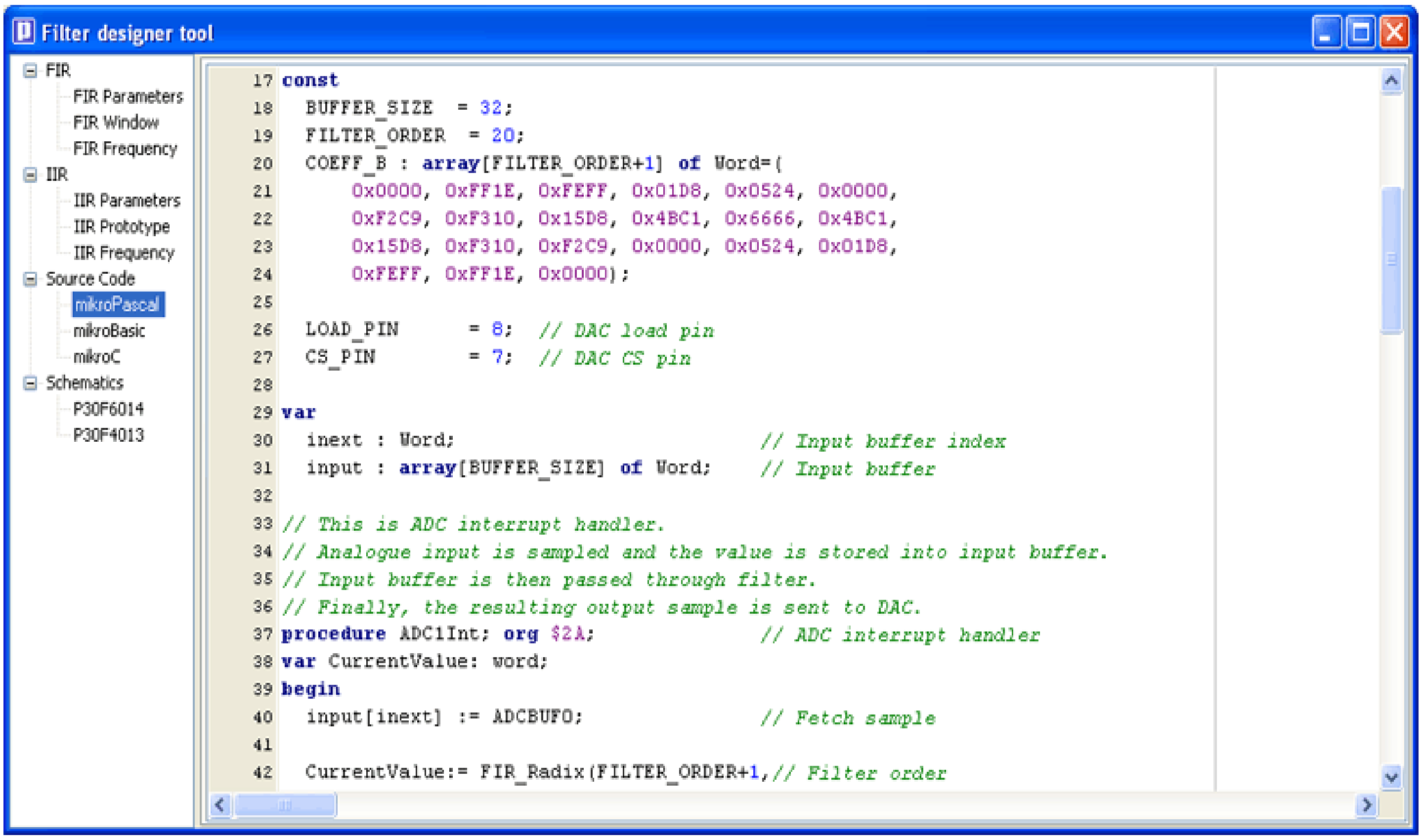


Fig. 15-8 Frequency characteristics of the window function (upper) and designed FIR filter (lower)

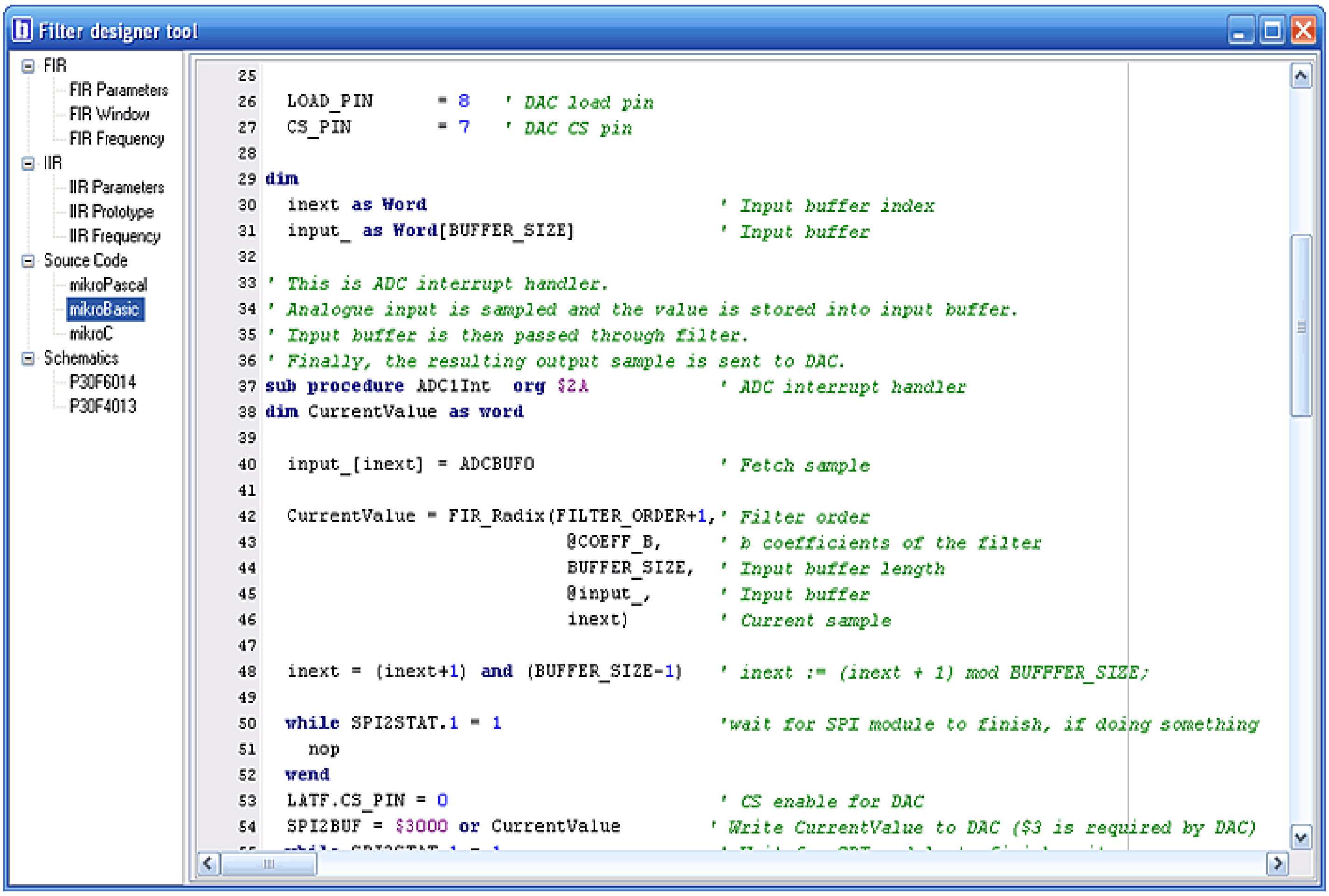
In the frequency characteristic of the FIR filter one can observe the deviation due to the finite wordlength (the coefficients are kept as 16-bit values). For the designed filter this deviation is negligible, but it can become significant for a higher order filter.

The final product is the code in Pascal, Basic or C which could be accessed by selecting mikroPascal, mikroBasic or mikroC in the left-hand part of **Filter Designer Tool**. The corresponding parts of the code are shown in Figs. 15-9, 15-10. and 15-11.



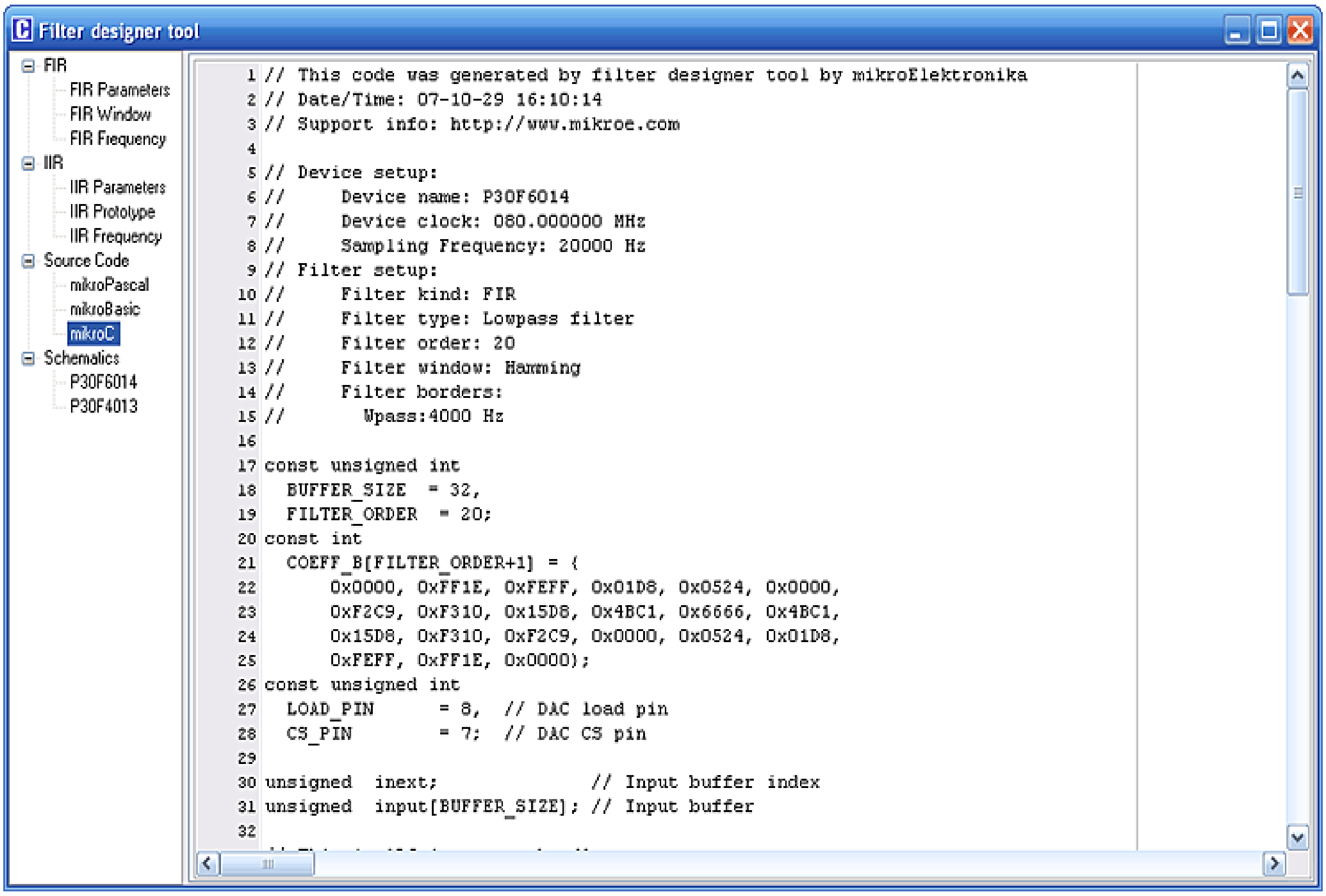
```
const
  BUFFER_SIZE  = 32;
  FILTER_ORDER  = 20;
  COEFF_B : array[FILTER_ORDER+1] of Word=(
      0x0000, 0xFF1E, 0xFEFF, 0x01D8, 0x0524, 0x0000,
      0xF2C9, 0xF310, 0x15D8, 0x4BC1, 0x6666, 0x4BC1,
      0x15D8, 0xF310, 0xF2C9, 0x0000, 0x0524, 0x01D8,
      0xFEFF, 0xFF1E, 0x0000);

  LOAD_PIN      = 8;  // DAC load pin
  CS_PIN        = 7;  // DAC CS pin

var
  inext : Word;                        // Input buffer index
  input : array[BUFFER_SIZE] of Word;  // Input buffer

// This is ADC interrupt handler.
// Analogue input is sampled and the value is stored into input buffer.
// Input buffer is then passed through filter.
// Finally, the resulting output sample is sent to DAC.
procedure ADC1Int; org $2A;            // ADC interrupt handler
var CurrentValue: word;
begin
  input[inext] := ADCBUF0;             // Fetch sample

  CurrentValue:= FIR_Radix(FILTER_ORDER+1,// Filter order
```

Fig. 15-9 Program code for designing FIR filter (mikroPascal)



```

  LOAD_PIN      = 8   ' DAC load pin
  CS_PIN        = 7   ' DAC CS pin

dim
  inext as Word                        ' Input buffer index
  input_ as Word[BUFFER_SIZE]          ' Input buffer

' This is ADC interrupt handler.
' Analogue input is sampled and the value is stored into input buffer.
' Input buffer is then passed through filter.
' Finally, the resulting output sample is sent to DAC.
sub procedure ADC1Int  org $2A         ' ADC interrupt handler
dim CurrentValue as word

  input_[inext] = ADCBUF0              ' Fetch sample

  CurrentValue = FIR_Radix(FILTER_ORDER+1,' Filter order
                           @COEFF_B,     ' b coefficients of the filter
                           BUFFER_SIZE,  ' Input buffer length
                           @input_,      ' Input buffer
                           inext)        ' Current sample

  inext = (inext+1) and (BUFFER_SIZE-1)  ' inext := (inext + 1) mod BUFFFER_SIZE;

  while SPI2STAT.1 = 1                 'wait for SPI module to finish, if doing something
    nop
  wend
  LATF.CS_PIN = 0                      ' CS enable for DAC
  SPI2BUF = $3000 or CurrentValue     ' Write CurrentValue to DAC ($3 is required by DAC)
```

Fig. 15-10 Program code for designing FIR filter (mikroBasic)



```
// This code was generated by filter designer tool by mikroElektronika
// Date/Time: 07-10-29 16:10:14
// Support info: http://www.mikroe.com

// Device setup:
//     Device name: P30F6014
//     Device clock: 080.000000 MHz
//     Sampling Frequency: 20000 Hz
// Filter setup:
//     Filter kind: FIR
//     Filter type: Lowpass filter
//     Filter order: 20
//     Filter window: Hamming
//     Filter borders:
//       Wpass:4000 Hz

const unsigned int
  BUFFER_SIZE  = 32,
  FILTER_ORDER  = 20;
const int
  COEFF_B[FILTER_ORDER+1] = {
      0x0000, 0xFF1E, 0xFEFF, 0x01D8, 0x0524, 0x0000,
      0xF2C9, 0xF310, 0x15D8, 0x4BC1, 0x6666, 0x4BC1,
      0x15D8, 0xF310, 0xF2C9, 0x0000, 0x0524, 0x01D8,
      0xFEFF, 0xFF1E, 0x0000};
const unsigned int
  LOAD_PIN      = 8,  // DAC load pin
  CS_PIN        = 7;  // DAC CS pin

unsigned  inext;              // Input buffer index
unsigned  input[BUFFER_SIZE]; // Input buffer
```

Fig. 15-11 Program code for designing FIR filter (mikroC)

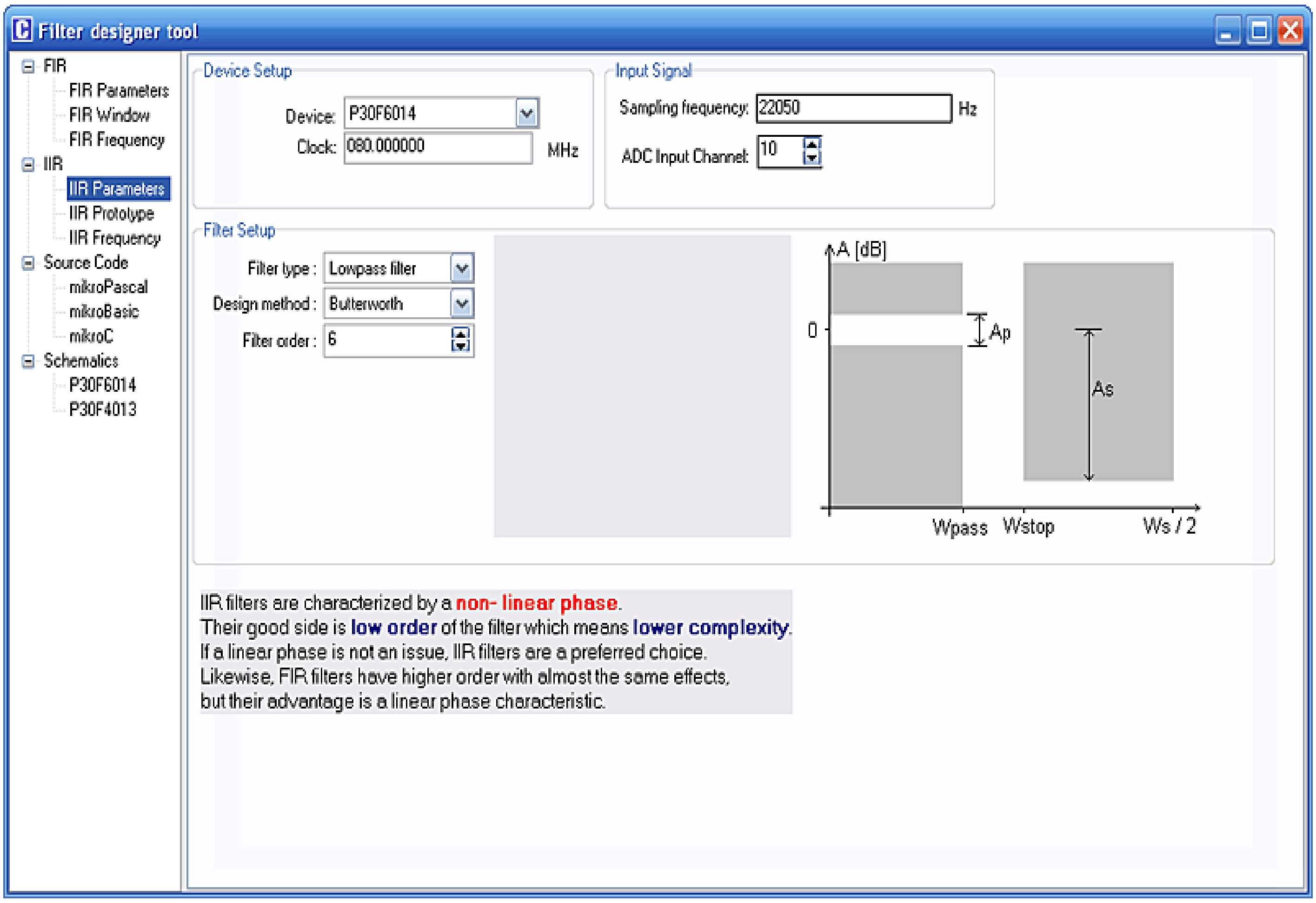
15.3 IIR filters

IIR is the abbreviated notation for the infinite impulse response digital filters. The basic properties of IIR filters are:

1. low order,
2. nonlinear phase,
3. potential instability.

Attention!!!
When you are using IIR filters with order > 3, it is important to know that they are vary sensitive to wordlength effects and they can become unstable. IIR filters with order > 3, are normally broken down into smaller sections, typically second and/or first order blocks, which are then connected up in cascade or in parallel. For details refer to : "Signal Processing : Discrete Spectral Analysis, Detection and Estimation" - M.Schwartz and L.Show (publisher MCGraw-Hill). "Digital Signal Processing" - E.C.Ifeachor and B.W.Jervis (publisher Addison-Wesley) Keep in mind that our filter designer tools implements IIR filters in direct canonical form, so you should not use IIR filters with order >3.

Design of an IIR filter starts by selecting Filter Designer Tool from the main menu of the mikroC compiler for dsPIC. The option IIR parameters on the left-hand side of the menu is selected. The form shown in Fig. 15-12 appears.



Device Setup

Device: P30F6014

Clock: 080.000000 MHz

Input Signal

Sampling frequency: 22050 Hz

ADC Input Channel: 10

Filter Setup

Filter type : Lowpass filter

Design method : Butterworth

Filter order : 6

IIR filters are characterized by a non- linear phase.
Their good side is low order of the filter which means lower complexity.
If a linear phase is not an issue, IIR filters are a preferred choice.
Likewise, FIR filters have higher order with almost the same effects,
but their advantage is a linear phase characteristic.

Fig. 15-12 Entering IIR filter parameters

In the part Device setup appear the clock and designation of the used microcontroller.

In the part Input Signal, two parameters, sampling frequency and input channel of the AD converter should be selected. The sampling frequency has to be at least twice as high as the maximum frequency of the input signal. If, e.g. the input signal is in the range 230Hz – 7500Hz, the minimum sampling frequency is 15kHz. Of course, one never selects the minimum frequency, thus in this case the sampling frequency of at least 20kHz should be selected.

In the part Filter settings the type of filter is selected first:

- Lowpass filter,
- Highpass filter,
- Bandpass filter,
- Bandstop filter.

Then, the order of the filter is selected. As the order of the filter increases, the selectivity of the filter will be higher (narrower transition zone), but the complexity of the filter will increase as a consequence of the more memory required for storing the samples and longer processing. The maximum sampling frequency depends on the length of processing (filter order) and selected clock.

After the type of IIR filter is selected, the method of IIR filter design can be selected, as shown in Fig. 15-13.

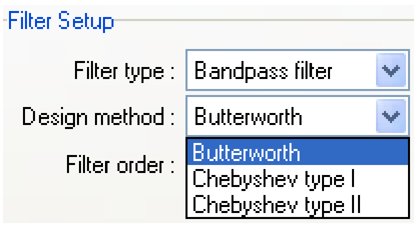
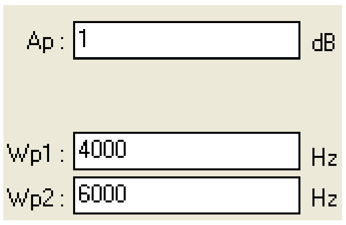

Filter Setup
Filter type : Bandpass filter
Design method : Butterworth
Filter order : Butterworth
Chebyshev type I
Chebyshev type II

Fig. 15-13 Selecting IIR filter design method

The basic IIR filter design methods are:

1. Butterworth – Maximally flat amplitude characteristic at 0Hz frequency. The amplitude characteristc is a smooth curve.
2. Chebyshev I – The minimum uniform ripple of the amplitude characteristc in the passband.
3. Chebyshev II – Known as the Inverse Chebyshev filter. The amplitude characteristc is maximally flat at 0Hz (like Butterworth filter). The least variation of the amplitude characteristc in the bandstop range.

Depending on the selected design method and type of IIR filter, it is possible to enter the corresponding parameters, whereas the remaining filter parameters will be calculated automatically in the sourse of filter calculation. If the Bandpass filter and Chebyshev I method are selected, the parameters to be entered are: the maximum attenuation in the bandpass range Ap and bandpass boundary frequencies Wp1 and Wp2.

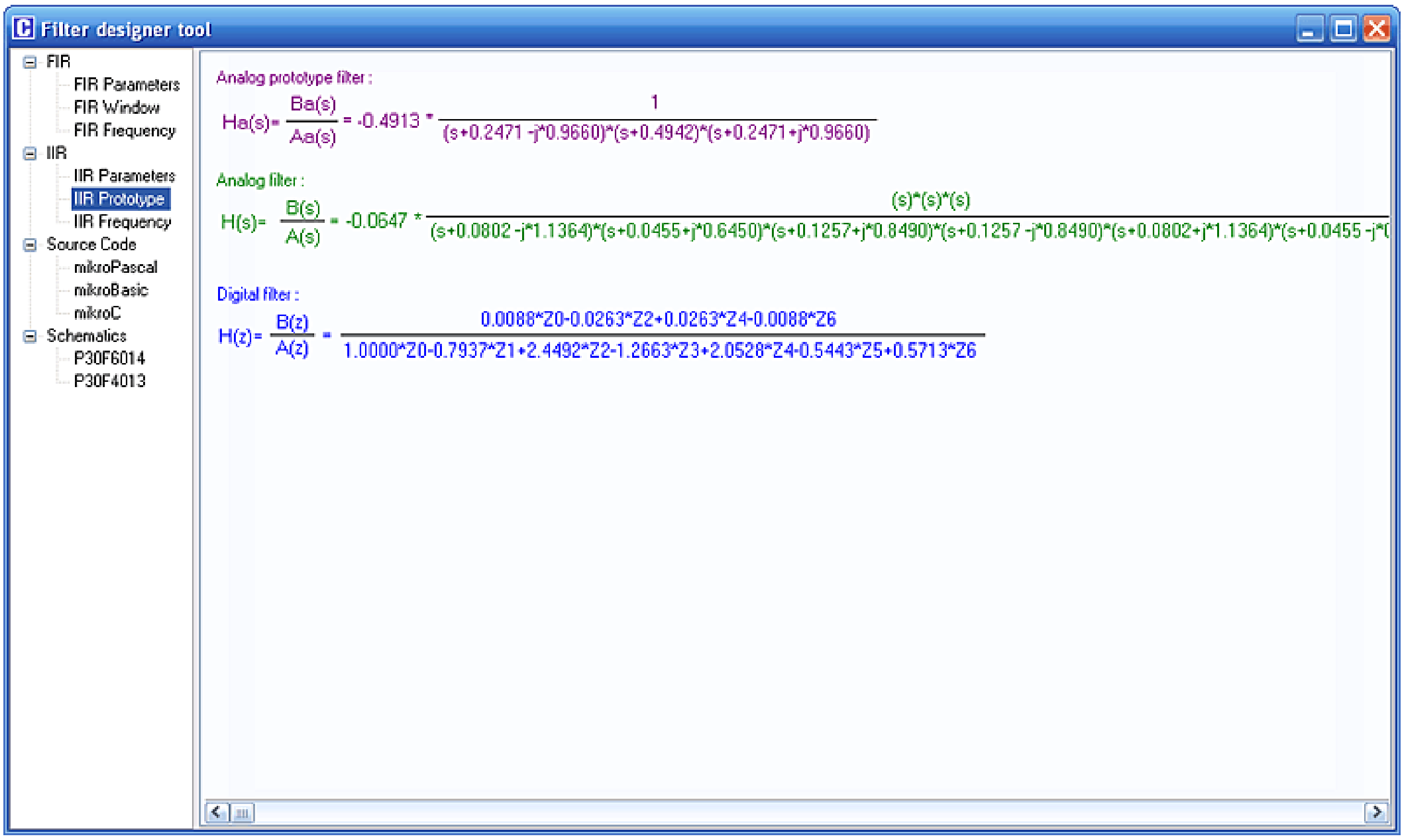

Ap : 1 dB

Wp1 : 4000 Hz
Wp2 : 6000 Hz

Fig. 15-14 Entering the bandpass attenuation and boundary frequencies

When all IIR filter parameters have been entered, design of the filter is carried out simply by selecting one of the options on the left-hand side of **Filter Designer Tool** (IIR Prototype, IIR Frequency, mikroPascal, mikroBasic, mikroC).

In the part **IIR Prototype** one can see the transfer functions of the analogue filter prototype, analogue filter, and digital filter. The analogue prototype filter and analogue filter are the transfer functions obtained in the course of calculations of the characteristcs of the digital filter. For a set of selected parameters, one obtains transfer functions as shown in Fig. 15-15.



Analog prototype filter :

$$Ha(s) = \frac{Ba(s)}{Aa(s)} = -0.4913 * \frac{1}{(s+0.2471 -j*0.9660)*(s+0.4942)*(s+0.2471+j*0.9660)}$$

Analog filter :

$$H(s) = \frac{B(s)}{A(s)} = -0.0647 * \frac{(s)*(s)*(s)}{(s+0.0802 -j*1.1364)*(s+0.0455+j*0.6450)*(s+0.1257+j*0.8490)*(s+0.1257 -j*0.8490)*(s+0.0802+j*1.1364)*(s+0.0455 -j*}$$

Digital filter :

$$H(z) = \frac{B(z)}{A(z)} = \frac{0.0088*Z0-0.0263*Z2+0.0263*Z4-0.0088*Z6}{1.0000*Z0-0.7937*Z1+2.4492*Z2-1.2663*Z3+2.0528*Z4-0.5443*Z5+0.5713*Z6}$$

Fig. 15-15 Transfer functions of the analogue filter prototype, analogue filter, and obtained digital filter

In the part **IIR Frequency** one can see the amplitude characteristic of the obtained IIR filter. The present case is shown in Figs. 15-16 and 15-17.

The final product is the code in Pascal, Basic or C which could be accessed by selecting mikroPascal, mikroBasic or mikroC in the left-hand part of **Filter Designer Tool**.

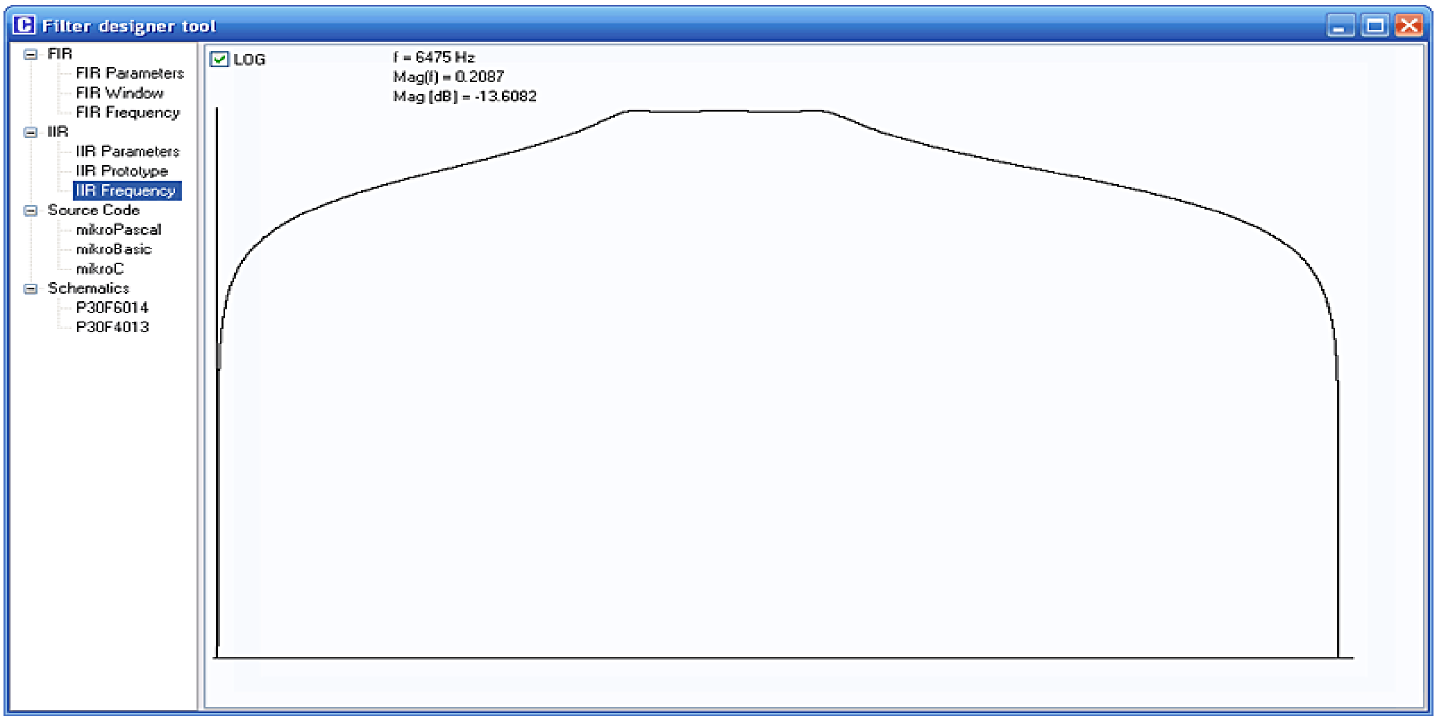
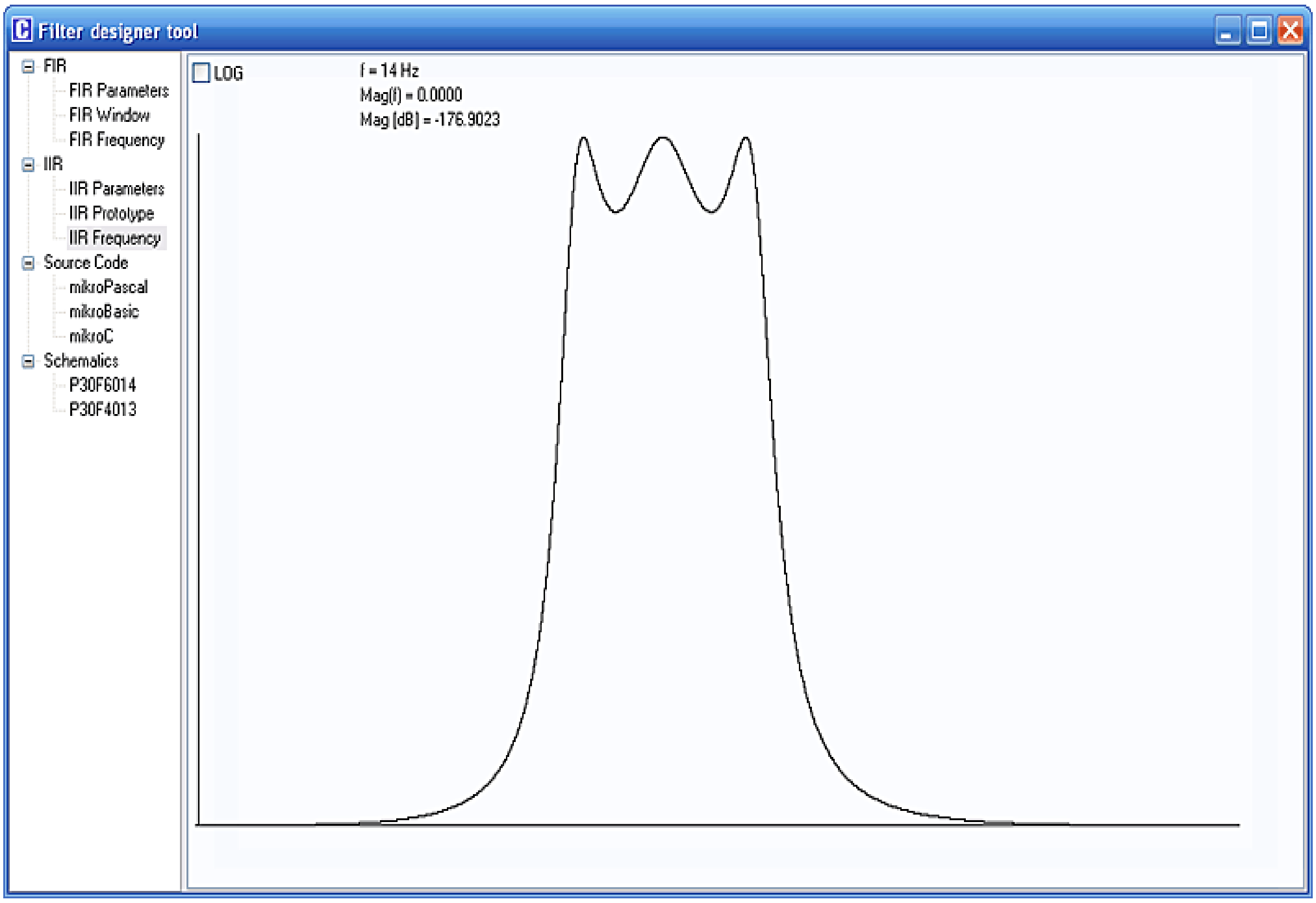


Fig. 15-16 Amplitude characteristic of the obtained IIR filter in LOG scale



f = 14 Hz
Mag(f) = 0.0000
Mag [dB] = -176.9023

Fig. 15-17 Amplitude characteristic of the obtained IIR filter in LIN scale