# Laboratory practical with the C8051Fxxx microcontroller family

Gingl Zoltán, Mingesz Róbert



**2014**

A tananyag a TÁMOP-4.1.2.A/1-11/1-2011-0104 "A felsőfokú informatikai oktatás minőségének fejlesztése, modernizációja" c. projekt keretében a Pannon Egyetem és a Szegedi Tudományegyetem együttműködésében készült.


# *Laboratory practicals with the C8051Fxxx microcontroller family*

Authors:

Dr. Zoltán Gingl and Dr. Róbert Zoltán Mingesz

**Keywords:**

Microcontrollers, embedded programming, timers, counters, serial communication, analogue-to-digital conversion, sensors.

**Summary**

The purpose of this book is to help the teaching of the applications of microcontrollers in various projects. Several books and manuals are available [1-19]; this book contributes to these by covering the knowledge needed to use the powerful C8051Fxxx family of microcontrollers from Silicon Laboratories in practice. Our aim was to synthesise the most useful information found in manuals, tutorials, datasheets, user forums, application notes, electronic design notes and example code in a single book. Most chapters feature brief application guidelines and troubleshooting based on our teaching and development experience. This can be highly useful for students and for developers as well.

We believe that the brief discussion of the architecture, peripherals, analogue and digital signal interfacing helps to understand how these can be used to build various applications. We provide tested example code and recommended exercises and discuss several application examples, including single-supply analogue signal conditioning, sensor interfacing and microcontroller-host computer communication. In the last chapter, we show the schematic and layout of an extension board that supports the use of the C8051F410DK development kit and can also be modified for use with other target boards.

Up-to-date, high quality references were chosen that are provided by industry leading companies [1–19]. Almost all of the references are available on-line on the companies' web pages.

TABLE OF CONTENTS

# 1 Introduction

## 1.1 Real-world signal processing and control

It is a typical aim to construct machines to make life more comfortable and more economical. From simple mechanical machines to advanced electronic devices such as smart phones the range is really wide. The most efficient devices are based on electronics, sophisticated signal processing and modern software.

In order to allow processing, real signals must be converted into another format that can be processed and the result should be used for intervention, as shown in Figure 1.1.

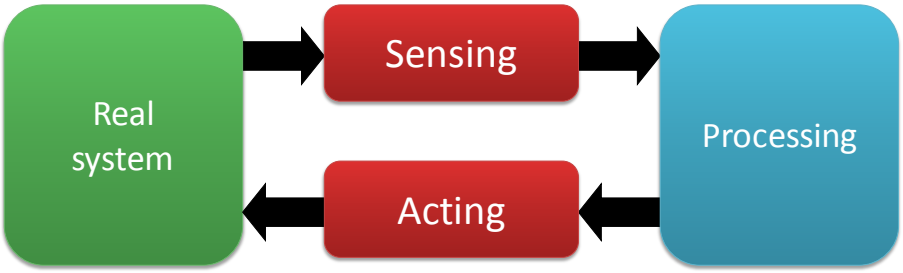


*Figure 1.1. General real-world interaction.*

The same principle is used in machines in general (Figure 1.2).

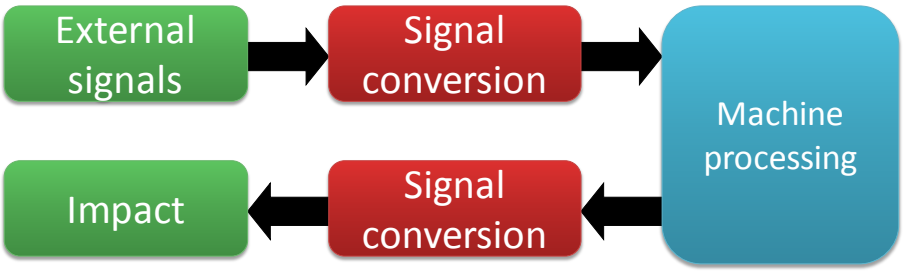


*Figure 1.2. Machine – real world interaction.*

The most efficient devices use analogue and digital electronics and run software to process information. Many of today's devices are small, battery-operated and incredibly efficient. Again, a good example is the smart phone that integrates telephony, camera, wireless communication, computer, sensors, GPS and many more in a handful of electronics.

The detailed block diagram of such an electronic device is shown in Figure 1.3. Sensors convert several physical signals (displacement, force, pressure, acceleration, temperature, light intensity, etc.) to signals that can be handled by electronics (voltage, current, resistance, capacitance, inductance). The output of sensors is converted to voltage in the proper range (a few volts) that can be easily used in processing. The analogue-to-digital converter translates this voltage to integer numbers for digital processing. A similar principle is applied in the reverse transformations.

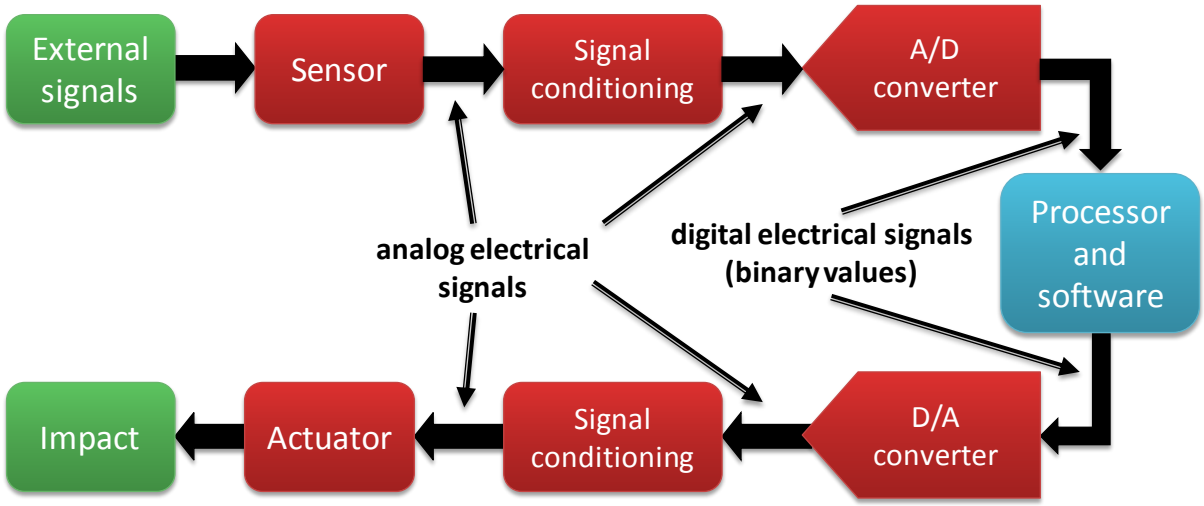


*Figure 1.3. Electronic device – real world interaction.*

Several analogue and digital integrated circuits have been developed to support the manufacture of electronic devices. One of the most compact and most efficient components is the microcontroller.

## *1.2 Microcontrollers*

The microcontroller unit (MCU) is a small but powerful digital building block, a single-chip microcomputer. It contains everything required for operation; very few external components are needed – sometimes only supply decoupling capacitors. Of course, the device must be powered, typically from a single supply voltage that ranges from 1.8 V to 5 V. Sometimes even a coin cell battery suffices.

The microcontroller has several peripherals to sense real-world signals and initiate real-world events, and has a processor core to run software. It is a very flexible, powerful and compact electronic component. Since most of the processing is done by the software, the same hardware can be used for several applications; the performance can be upgraded easily by replacing the software only.

There is a very wide range of microcontrollers on the market from sizes of 2 mm × 2 mm and from a power consumption of 30 μW to a speed of several hundred MHz.

Most modern microcontrollers incorporate comparators, analogue-to-digital and digital-to-analogue converters and temperature sensors – therefore, they are often called mixed-signal (both analogue and digital) microcontrollers.

Figure 1.4 illustrates some typical components of a modern mixed-signal microcontroller; the details will be given in the next chapter.

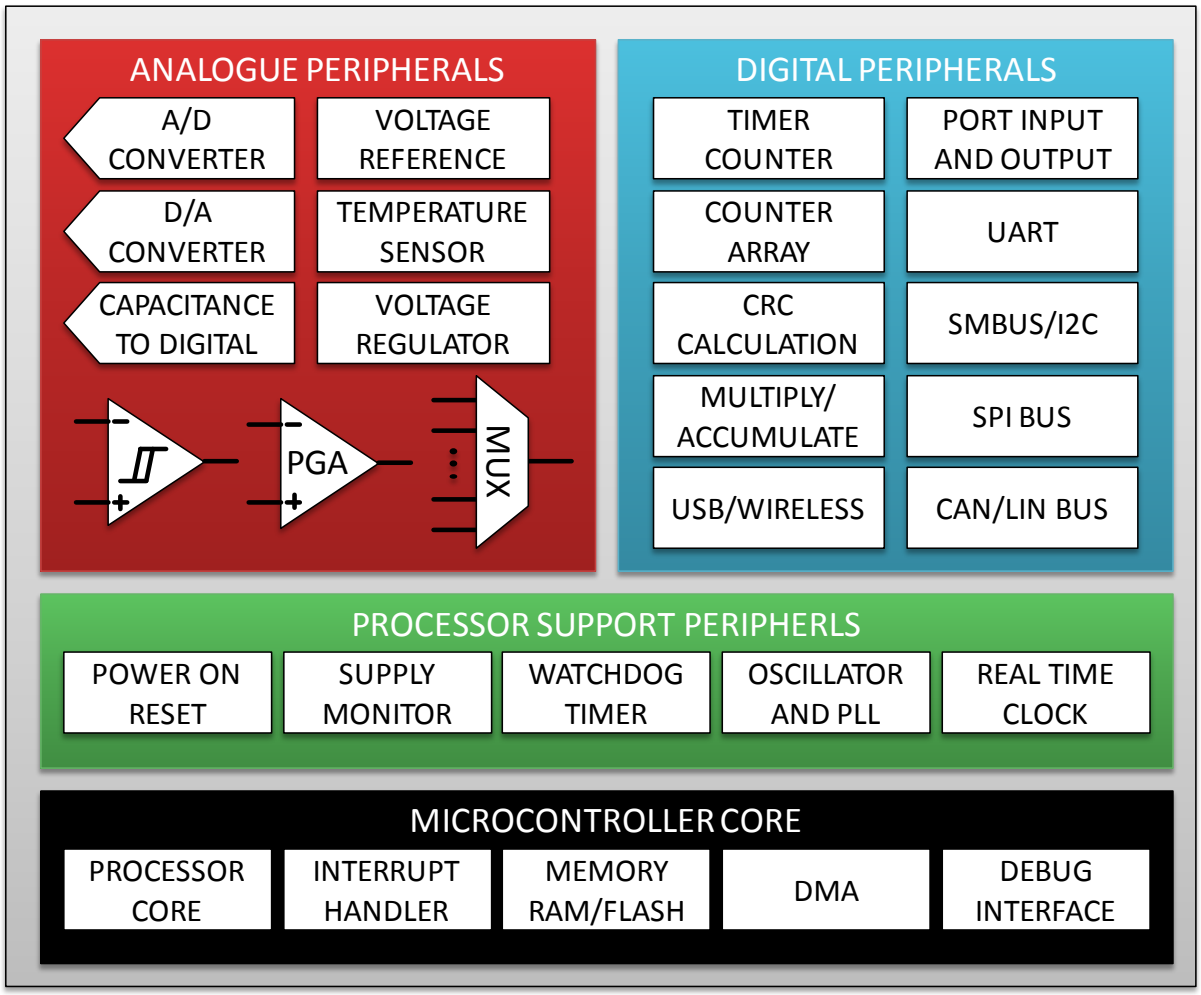


*Figure 1.4. Microcontroller components.*

## *1.3 Microcontroller core and integrated peripherals*

The microcontroller core is based on a processor with its typical components including an arithmetic logic unit (ALU) and several registers. The architecture may follow the CISC (like the 8051 family) or, more probably, the RISC principles (for example, the PIC, AVR and ARM microcontrollers) in today's popular microcontrollers.

Most of the devices use separate memory for the data and for the program; that is, they have Harvard architecture. This fits well the need for non-volatile program memory and at the same time it prevents code corruption and provides even faster execution in some cases. The word length of the two kinds of memory can also be different. Microcontrollers may use Neumann or Harvard architecture, or the user can even configure the memory usage (for example, in the case of the ARM Cortex-M3 32-bit microcontroller family).

All modern microcontrollers have volatile (SRAM) memory and non-volatile, reprogrammable flash memory. The flash memory contains the code, so no external integrated circuits are needed. The flash memory can be reprogrammed by special programming devices using a few (from 2 to 6-8) pins of the microcontroller (in-circuit programming, JTAG) or can even be overwritten by the microcontroller itself in some cases. Additional separate flash or EEPROM may also be integrated to support non-volatile data storage (configuration data, calibration data, statistical data, etc.). The flash memory can be

rewritten about 100000 times, and the typical data retention time is longer than 20 years. The flash memory can be protected, i.e., the code can be prevented from being read by the user.

If the on-chip memory is not enough for a certain application, the developer can choose microcontrollers with an external memory interface that support the connection of static RAM or other memories of various sizes. Note also that this interface may support the use of 'memory mapped' peripherals including A/D converters, D/A converters, FIFO memories, etc.

### *1.3.1 Processor support*

In the following the most typical processor support peripherals will be described briefly.

**Power on reset (POR)** generator. After switching the power on, the supply voltage may rise a bit slowly due to the fact that the supply decoupling and filtering capacitors must be charged and the supply current is limited. At the same time, the digital circuitry needs a certain minimum supply voltage for proper operation, so the start-up of the microcontroller must be delayed until the supply voltage reaches the safe operating level. Having detected the crossing of this level, the POR generates an additional short delay (in the range from below 1 ms to about 100 ms) and finally releases the reset line.

**Power supply monitor** (Brown-out detector). In some cases, the supply voltage may go below the safe operating level even during operation (for example, when sudden heavy current loading occurs). This may result in erroneous code execution, therefore the supply monitor circuit will generate a reset in this case. Note that this feature can be disabled by the programmer, although the use of the supply monitor is strongly recommended.

**Low-dropout (LDO) regulator.** Some microcontrollers have separate voltage levels for their core and digital input and output ports. Integrated voltage regulators can provide stable and sometimes even programmable supply voltage from the input supply voltage. Low-dropout regulators need only a slightly (roughly about 100 mV) higher input supply voltage than their output voltage.

**Watchdog timer (WDT)**. Even properly powered processors can fall into infinite loops or get disturbed by electromagnetic or conducted interference (for example, in the case of lightning or power line transients), which may cause serious problems in several applications (motor control, heating control, healthcare devices, etc.). The watchdog timer refresh register needs to be written within a certain amount of time (that can be typically programmed from tens of milliseconds to several seconds); otherwise, a reset will be generated. If the code writes to the register, the timer will be restarted and no reset will be generated. If the processor code execution fails, this will not occur and a reset will be initiated. The best practice is to always use the watchdog timer except in code development phase or in simple test projects. The watchdog timer is enabled automatically upon reset in quality microcontrollers.

**Oscillator, PLL**. All processors need a clock signal to schedule instruction execution. Modern microcontrollers have on-chip oscillators but also support the use of external quartz crystals or external clock signals. Optional phase-locked loop (PLL) clock multipliers often combined with clock dividers allow the generation of a wide range of higher processor clock frequencies. Typically, on-chip oscillators have an accuracy of 1%-20%, while the precision of

crystal oscillators can fall below 0.01%. The developer can choose the solution that suits the particular application best.

**Debug interface.** This interface is used by the integrated development environment to download code to the flash memory. Memory upload is also supported and the developer can program the security bits to protect the code from being uploaded. The debug port allows single stepping, supports breakpoints and can track the content of variables, memory and peripheral registers. The debug interface makes code development and testing easy and it is an essential tool for all modern microcontrollers. The most commonly used interface standard is called JTAG (Joint Test Action Group, IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture).

### *1.3.2 Digital peripherals*

Digital peripherals include the digital input/output pin drivers and internal digital circuits related to timing, communication and computation acceleration.

**General-purpose input/output (GPIO), port input/output (Port I/O).** The processor reads from and writes to memory and all on-chip peripherals using the bidirectional data bus. Some processors may also incorporate a direct memory access (DMA) controller, which transfers data between memory and a peripheral without processor intervention. The data is valid only for a short duration in order to free the bus for other transactions, so the general purpose output requires latches that can keep the data until the code writes new data to it. The output of these latches is connected to the pins of the chip and can drive LEDs and provide logic output signals for external digital circuit inputs. These signals are mostly arranged in 8-bit groups to form a byte. The pins can also be configured as digital inputs that can be read any time by the microcontroller. This way buttons, switches and digital signals can be connected as well with the help of internal or external pull-up resistors.

**Timer/Counter modules.** Microcontrollers are designed to control electronic equipment for household, automotive, industrial, test and measurement applications; therefore, timing, event counting, periodic event generation and time duration measurement are important. All microcontrollers contain 8-, 16- or 32-bit counters that can be configured as timers (when an oscillator drives the counter) or as counters, when the rising or the falling edge of an external signal increments the counter. Timers also provide timing for serial communication peripherals, A/D converters and D/A converters.

**Programmable Counter Array (PCA).** The PCA contains a simple free-running counter that is driven by an oscillator. There are several (from 3 to 6) independent compare/capture registers that can be used to latch the counter value upon an event (a change in a digital input signal). These registers can also hold data to be compared with the counter value and to generate an event when a match occurs. The PCA can be used to measure pulse width, period or frequency, to generate pulse width modulated (PWM) signals and special logic signal patterns, periodic interrupts and even more.

**Real-Time clock (RTC).** In order to measure the real time or synchronise events to it, a dedicated precise oscillator and an associated 32 to 48-bit counter is provided in some microcontrollers. The oscillator typically uses 32768-Hz tuning fork crystals and a very low power oscillator. Practically, a clock is integrated into the microcontroller that can be powered from a button battery and can run even if the processor is not powered. Besides

measuring real time, this peripheral can serve to wake the microcontroller up at a certain time – in other words, to provide alarm function.

**Computing support (MAC, CRC, AES)**. Some microcontrollers contain computation and digital signal processing acceleration hardware. For example, 8-bit microcontrollers can have a multiply and accumulate (MAC) unit that can multiply and add 16-bit data in a few clock cycles. This can be used efficiently in digital filtering and to compute fast Fourier transforms (FFT). Cyclic redundancy check (CRC) is frequently used to check data integrity in communications and the Advanced Encryption Standard (AES) algorithm is also supported by some microcontrollers.

### *1.3.3 Communication*

**Universal Asynchronous Receiver/Transmitter (UART).** This serial (one bit at a time) communication interface uses one wire to send and another wire to receive bits of a byte. The sender and receiver must have a closely matched time base that determines the duration of transmitting a bit, since no timing synchronisation is provided. Every transaction is initiated by sending a start bit, followed by the data bits. The receiver detects the start bit and can then decode the data bits by sampling the signal at evenly spaced time instants. The UART interface is used for low wire count inter-processor communications, host computer communication via USB-UART interfaces, infrared communications and device-to-device communications. In most cases, it is a two-device bus; the use of more devices introduces hardware and software overheads.

**Serial Peripheral Interface (SPI).** The SPI interface is typically used for high-speed communication with off-chip peripherals including analogue-to-digital converters, digital-to-analogue converters, digital output sensors and other processors. Two wires are used to carry data bits in two directions and one wire for a clock signal that synchronises the timing between the communicating devices. A rising or a falling edge of this signal indicates the beginning of the transmission of each bit. A fourth signal may also be used to provide a frame for the communication. If this signal is inactive, the other signals are ignored, which can be used to connect multiple devices on the same bus and select one for which communication is enabled.

**Inter-Integrated Circuit (IIC or I²C) and System Management Bus (SMBus).** This medium-speed interface is specially developed for communication between a host microcontroller and several peripheral chips (memories, data converters, sensors or other processors) on the same printed circuit board or within equipment over only two wires. One wire carries data in both directions, while the other is used to provide a frame (start and stop conditions) for the transaction and to synchronise the transmission of the bits through clock pulses.

**Controller Area Network (CAN), Local Interconnect Network (LIN).** These serial interfaces are only available on some microcontrollers that target automotive or other industrial applications. Most of the protocol is implemented in hardware.

**Universal Serial Bus (USB).** The USB is the most popular and innovative interface for connecting peripherals to personal computers or tablets. Some microcontrollers have built-in slave (and rarely host) USB ports. This allows direct connection to the USB port; however, the programmer should know the most important parts of the USB protocol and a driver is typically required on the host computer.

**Wireless communication peripherals.** Wireless communication is becoming a more and more popular interface between small devices, since it supports very flexible location and networking options and no wires are required. There are microcontrollers with integrated wireless transmitters and receivers (transceivers) with several frequency options and a number of wireless protocols can be implemented by software. Bluetooth, ZigBee and the open-source TinyOS system are among the most widely used platforms.

### *1.3.4 Analogue peripherals*

Several microcontrollers have analogue parts to handle analogue signals even without external analogue circuitry. This makes microcontrollers even more compact: a single microcontroller and only a few external components can implement a complete solution for a real-world application that requires the monitoring of signals and the controlling of processes. Microcontrollers that have a significant analogue part and can therefore handle both digital and analogue signals are often called mixed-signal microcontrollers or analogue microcontrollers.

**Comparator.** Comparators have two analogue voltage inputs and a digital output. Their output is logical high if the voltage connected to their positive input is higher than the voltage at their negative input. They may also have hysteresis to reduce potential noise-induced switching.

**Analogue-to-Digital Converter (ADC).** Analogue voltages can be translated into the digital domain using ADCs. The output is an integer number with a various number of bits. A resolution of 10 bits is the most typical, but precision microcontrollers can have 12-, 16- or even 24-bit ADCs. Note that the accuracy is normally less than the resolution; therefore, the datasheet should always be consulted to obtain reliable information.

**Digital-to-Analogue Converter (DAC).** DACs output analogue signals proportional to the integer number at their input. The resolution range includes 8, 10, 12 or 16 bits. The output signal can be voltage or current.

**Voltage reference (VREF).** All data converters (ADCs, DACs) need a reference voltage that serves as an etalon of conversion. The input range of the ADCs and the output range of voltage output DACs are both determined by $V_{ref}$; in most cases it is between 0 and $V_{ref}$. The internal reference voltage can be switched off to support the use of more precise external reference voltage circuits.

**Capacitance-to-Digital Converter (CDC).** One of the most popular modern user interfaces is based on touch sensing that effectively replaces mechanical buttons, which have limited reliability and lifetime. The change in a capacitance is measured, which change depends on the proximity of the finger of the user from the sensing pad. The capacitance is digitised and the data can be used for evaluation.

**Analogue Multiplexer (MUX).** Monitoring multiple analogue signals is often needed in real-world applications. This can be supported by a network of switches, called an analogue multiplexer, that connects one of the signals to the input of the ADC at a time. After the conversion of a signal, the next signal can be selected. Since conversion only takes a short time, this means a quasi-simultaneous conversion if the signals change only slowly. However, the different signals are measured at slightly different time instants, which should be considered anyway.

**Programmable Gain Amplifier (PGA).** Some microcontrollers have preamplifiers before their integrated ADCs to support voltage range extension. The preamplifiers can have software-programmable gains of 0.5, 1, 2, 4, 8, 16, 32, 64, 128. Single-ended and differential input PGAs are both available.

**Temperature sensor.** Most mixed-signal microcontrollers include diode-based temperature sensors that can be connected to the input of the internal ADC using the analogue multiplexer. The on-chip sensor outputs a voltage that has linear dependence on the chip temperature. The accuracy of the sensor is roughly about ±3 °C. It can be used to protect the device from overheating or to estimate the ambient temperature if the power dissipation of the microcontroller is low enough so that we can neglect self-heating.

## *1.4 Microcontroller classification*

Depending on the different features and according to target applications microcontrollers can be broken down into the following categories:

**General-purpose microcontrollers** have common digital peripherals including timers, GPIO or UART. Their typical clock frequency is around 10 MHz.

**Low power microcontrollers** can operate at lower clock frequencies, from 1 MHz down to tuning fork crystal frequency of 32768 Hz or even below. At 1 MHz the supply current is well below 1 mA, and supply sensitivity is close to 200 μA/MHz. During power down state the supply current can fall below 1 μA.

**Precision mixed-signal microcontrollers** incorporate 12-bit or higher resolution ADCs and DACs. Sigma-delta ADCs can even have a resolution of 24-bits and a PGA can provide software programmable gains in the range of 1 to 128.

**High-speed microcontrollers** execute most of their instructions within a single clock cycle and can operate at frequencies from about 25 MHz to several hundred MHz.

According to the bus width there are **8-bit, 16-bit and 32-bit microcontroller** families. 8-bit microcontrollers typically consume less power, while 32-bit microcontrollers have more processing power.

**Industrial and automotive microcontrollers** operate at a full industrial temperature range of -40 °C to 85 °C. The internal peripherals have stricter specifications to provide additional reliability under various conditions, and accuracy of the internal oscillator is better than 1% over the full operating temperature range. These microcontrollers typically have industrial or automotive communication peripherals like CAN buses or LIN buses.

**Secure microcontrollers** are used in security-sensitive applications including electronic banking and payment, application protection, communication and more. These microcontrollers offer protection of code and data, prevent reverse engineering, tampering, data monitoring and physical attacks. Hardware cryptographic modules, random number generators, fast data and code encryption are implemented to support secure applications.

# 2 Architecture and properties of the C8051Fxxx microcontroller family

C8051Fxxx microcontrollers developed by Silicon Laboratories [1, 2] are among the most powerful modern derivatives of the popular MCS-8051 MCU [2] introduced by Intel. A short summary of these devices follows.

## 2.1 8051 microcontrollers

The 8051 or MCS-51 family of 8-bit Harvard architecture microcontrollers were developed by Intel in the eighties for embedded applications. Their easily upgradable architecture proved successful, became a standard for many manufacturers and several derivatives are still popular on the market due to their ease of use and carefully designed peripheral handling.

The 8051 family can be easily programmed. There are many free and professional development tools, so the 8051 microcontrollers can be used by practiced experts, lecturers, students and hobbyists at the same time. Many source code examples are available to solve various problems and the manufacturers provide very useful application notes, knowledge base and user forums.

Manufacturers include Silicon Laboratories, Maxim/Dallas, Analog Devices, Atmel and NXP (formerly Philips).

Very wide ranges of speed, code and data memory size, analogue and digital peripherals, power requirement are provided by the C8051Fxxx family developed by Silicon Laboratories. The 1 MIPS peak performance of the original 8051 microcontroller has been upgraded up to 100 MIPS peak speed and the integrated flash memory, debug interface and very rich set of analogue and digital peripherals make the C8051Fxxx family a good choice for various applications.

The chips can have sizes of 2 mm × 2 mm (10 pins) to 16 mm × 16 mm (100 pins).

## 2.2 The C8051Fxxx microcontroller family

The maximum clock frequency of the C8051Fxxx microcontrollers is in the range of 25 MHz to 100 MHz. Slower clock speeds are allowed, practically down to DC, so no minimum is specified. The frequency of the internal oscillator is programmable, so the user can choose low power operation at low frequencies, while higher processing speeds can be achieved at the expense of higher power consumption. For example, the C8051F410 processor can be operated at 50 MHz, when the core supply current is about 15 mA, while at 32 kHz the device draws less than 20 μA from the supply rail, allowing long lasting operation from a battery.

The size of on-chip flash memory available varies from 2 kbyte to 128 kbyte, while the internal RAM can store 256 to 8448 bytes of data. The flash memory contains the code and may also be written by the code to support non-volatile data storage.

The C8051Fxxx microcontrollers can have up to six 16-bit timers and a programmable counter array with 6 independent channels. Some devices include a real-time clock with battery backup power option.

Communication peripherals include UARTs, I²C/SMBus, SPI, USB, CAN, LIN serial interfaces and the parallel external memory interface that also supports the connection of fast external ADCs, DACs and more.

From 6 to 64 GPIO pins are available with configurable output driving options (open-drain with or without internal pull-up and push-pull mode).

The C8051Fxxx family provides high-performance analogue peripherals. ADC resolutions from 10 to 12 bits with sample rates from 100 kHz to 200 kHz are common, while the C8051F06x devices incorporate two independent 1-MHz 16-bit ADCs, and the C8051F35x microcontroller has an 8-channel 24-bit ADC with programmable-gain amplifier to resolve sub-μV signals. Some devices have a 32-channel multiplexer before their ADCs, and DACs with resolutions of 8 to 12 bits are also available. The list of analogue peripherals may also include up to 3 comparators with programmable response time and hysteresis.

The company provides several development tools including a free integrated development environment that supports the use of the popular open-source Small Device C Compiler (SDCC). A configuration wizard application helps much in configuring the peripherals properly by generating even the source code (assembly or C).

Hardware development platforms are also available. There are simple and full-featured development kits for almost all C8051Fxxx processors.

## *2.3 The CIP-51 architecture*

The Silicon Laboratories C8051Fxxx microcontrollers have the so-called CIP-51 architecture [6]. The simplified block diagram is shown in Figure 2.1.

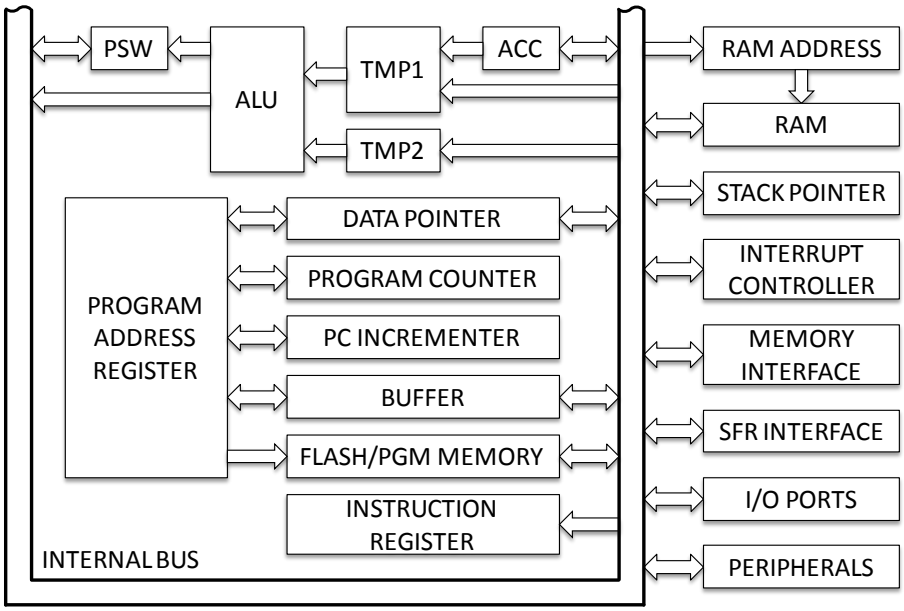


*Figure 2.1. A simplified CIP-51 architecture.*

The architecture is closely matched with the original 8051 architecture developed by Intel; code compatibility is provided. The main improvements include much faster instruction execution, integrated flash memory and larger integrated RAM.

In the following the main features of the architecture will be discussed.

### *2.3.1 Registers*

The following table summarises the 8-bit registers, with short descriptions and the reset values [2]. The registers can be used in several instructions. The accumulator (**A** or **ACC**)

holds the result of arithmetic and logic operations and the program status word (**PSW**), and contains several flags modified by operations. Additional registers support indirect addressing and stack handling. Instructions typically execute faster when the operands are registers.

| Register | Description | | | | | Reset value |
|---|---|---|---|---|---|---|
| **A, ACC** | accumulator, ALU result | | | | | 0 |
| **B** | general-purpose register and register for multiplication and division | | | | | 0 |
| **R0.–R7** | general-purpose registers, R0 and R1 are also used in indirect addressing | | | | | 0 |
| **PSW** | Bit 7: **CY** | carry bit (set by addition or subtraction, *ADDC, SUBB*) | | | | 0 |
| | Bit 6: **AC** | auxiliary carry bit (at 3rd bit, used in 4-bit arithmetics) | | | | 0 |
| | Bit 5: **F0** | user flag | | | | 0 |
| | Bit 4: **RS1** | R0–R7 at 00: 0x00 | R0–R7 at 01: 0x08 | R0–R7 at 10: 0x10 | R0–R7 at 11: 0x18 | 0 |
| | Bit 3: **RS0** | | | | | 0 |
| | Bit 2: **OV** | overflow (set by instructions *MUL, DIV, ADD, SUBB*) | | | | 0 |
| | Bit 1: **F1** | user flag | | | | 0 |
| | Bit 0: **PAR** | parity bit: 1 if sum of bits in A is 1 | | | | 0 |
| **DPH, DPL** | **DPTR**, data pointer, used in 16-bit indirect code or RAM addressing | | | | | 0 |
| **SP** | stack pointer, modified by subroutine and interrupt routine calls or push/pop instructions | | | | | 7 |

### *2.3.2 Special function registers*

The special function registers (SFRs) are used to access the peripherals and some registers. For example, **ACC** is the same as **A** (accumulator); therefore, it can be accessed as an SFR or as a register. This allows the accumulator to be used in some instructions when registers cannot be used (like *push* and *pop*, see later)

SFRs can be accessed by direct addressing instructions, where the address falls in the range of **0x80–0xFF**. Therefore, SFRs can be thought of as memory-mapped registers; the program can read or write their content as if they were in the RAM.

The following table shows the standard 8051 SFR registers.

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0xF8 | | | | | | | | |
| 0xF0 | B | | | | | | | |
| 0xE8 | | | | | | | | |
| 0xE0 | ACC | | | | | | | |
| 0xD8 | | | | | | | | |
| 0xD0 | PSW | | | | | | | |
| 0xC8 | | | | | | | | |
| 0xC0 | | | | | | | | |
| 0xB8 | IP | | | | | | | |
| 0xB0 | P3 | | | | | | | |
| 0xA8 | IE | | | | | | | |
| 0xA0 | P2 | | | | | | | |
| 0x98 | SCON | SBUF | | | | | | |
| 0x90 | P1 | | | | | | | |
| 0x88 | TCON | TMOD | TL0 | TH0 | TL1 | TH1 | | |
| 0x80 | P0 | SP | DPL | DPH | | | | |

Note that SFRs in column 0 are bit addressable.

The SFRs listed in the table are the following (some of them will be discussed in the next chapters):

- **P0, P1, P2** and **P3** are the port input/output SFRs that are associated with the pins of the microcontroller. For example, the byte written to **P0** determines the logic signal on the 8 pins corresponding to **P0**. The programmer must be careful: for example, writing 1 to **P0** sets the least significant bit but will clear all the other 7 bits. Since the **P0** register is bit addressable, a single bit can be written or read without affecting the other bits. For example, setting **P0.0** sets the least significant bit only; all the other bits remain unchanged. Bit addressing is also useful for accessing a single bit of the status and other registers where the individual bits have special meanings.
- **ACC** and **B** provide **SFR** access to the accumulator and to the B register.
- **PSW** is the program status word. Its individual bits are accessible using bit addressing. For example, **PSW.7** is the carry bit.
- **SP** is the stack pointer.
- **DPL** and **DPH** are the low- and high-order bytes of the data pointer **DPTR**.
- **IE** and **IP** are the interrupt enable and priority registers. Their individual bits are accessible using bit addressing.
- **TCON**, **TMOD**, **TL0**, **TH0**, **TL1** and **TH1** are used to access and control the Timer 0 and Timer 1 peripherals.

**SCON** and **SBUF** are associated with the serial port communication peripheral.

### *2.3.3 Memory structure*

8051 processors have Harvard architecture [2]; they have separate memory for code and data. The code memory can store constant data, so it can be used as a read-only data memory. Two types of RAM are available: internal and external. The internal RAM size is 256 bytes, while the external RAM is addressed by a 16-bit pointer, so the maximum size is 64 kbyte.

Figure 2.2 shows the internal RAM structure. The first 128 bytes (from 0x00-0x7F) can be accessed by direct or indirect addressing. The general-purpose registers occupy 8 bytes at the location defined by the **RS0** and **RS1** bits of the **PSW** register. The 16-byte space at address 0x20-0x2F is bit addressable, so 128 individual bit variables can be used here.

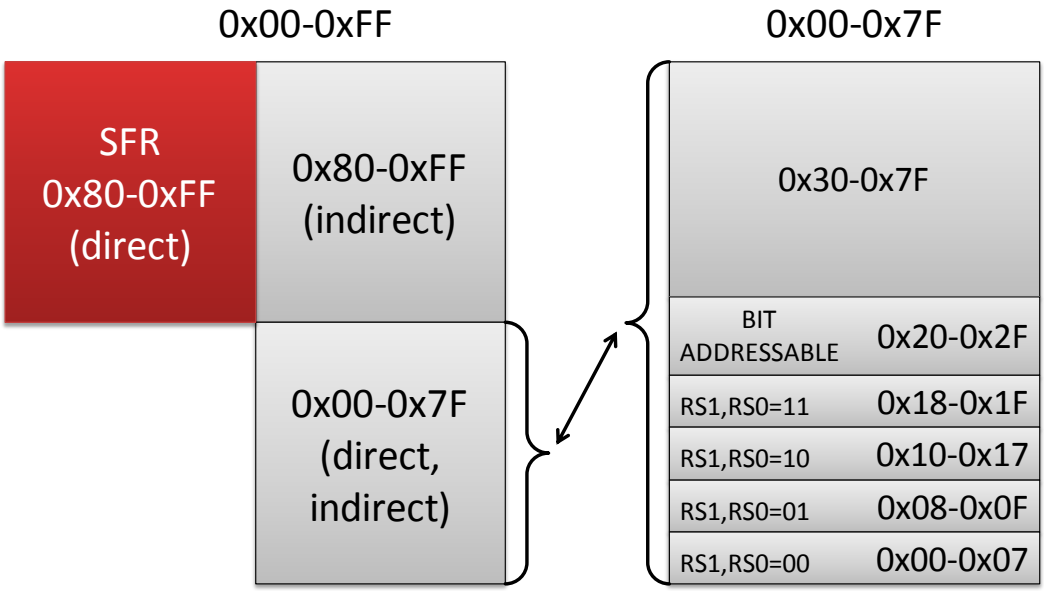


*Figure 2.2. Internal memory structure of CIP-51 microcontrollers.*

The SFR registers are mapped to the upper 128 bytes of the address space. SFRs are accessed by direct addressing; otherwise, the upper 128 bytes of the internal RAM can be used. Note that since stack handling is based on indirect addressing by the stack pointer, the upper 128 bytes of RAM can also be used as stack space. Upon reset, the stack pointer has the value of 7 and increases from there. However, it is best to set the initial value of the stack pointer (**SP**) to the first free location of data memory, just above the variables. In this case, all free memory is available as stack.

The external RAM (XRAM) was originally provided by SRAM chips, but modern C8051Fxxx processors integrate a certain amount (up to 8192 bytes) of this kind of RAM. XRAM memory can only be accessed by 16-bit indirect addressing using the **DPTR** pointer (**DPH** and **DPL** registers).

XRAM at 0x00-0xFF can also be accessed by 8-bit indirect addressing using either the **R0** or the **R1** register.

Since off-chip memory can be slower than the on-chip memory, the control timing (data setup and hold time, write/read pulse width, etc.) can be set by dedicated SFR registers.

Figure 2.3 shows the XRAM arrangement in C8051Fxxx processors. The processor can be configured to access the on-chip memory only, the off-chip memory only or on-chip only if it is available and off-chip otherwise. The 8-bit addressable space can also be moved to another 256-byte page. Note that not all C8051Fxxx processors support off-chip memory.

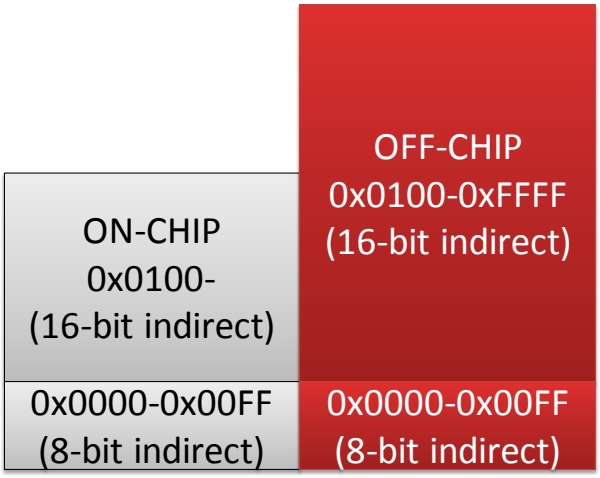


*Figure 2.3. External memory structure of CIP-51 microcontrollers.*

### 2.3.4 Addressing modes

Data can be accessed in different ways depending on its location (register, memory or code) and on the so-called addressing mode. The following table summarises the four possible addressing modes and shows examples.

| Addressing mode | MNEMONIC example | Description |
|---|---|---|
| ***register*** | MOV A, B | A = B, copy the content of B to A |
| ***immediate constant*** | MOV A, #10 | A = 10 (value), copy the value 10 to A |
| ***direct*** | MOV A, 10<br>MOV A, P0 | A = byte in internal RAM at address 10<br>A = bits at port P0 (SFR access) |
| ***indirect*** | MOV A, @R0<br>MOVX A,@DPTR | A = byte in internal RAM at address pointed to by R0<br>A = byte in external RAM at address pointed to by DPTR |

### 2.3.5 Instructions

A brief summary of the available instructions are given in the following [2]. Instructions are classified into groups and tables summarise their function and the flags affected by them.

### *2.3.5.1 Arithmetic operations*

| MNEMONIC | OPERATION (C-style syntax) | ADDRESSING | | | | FLAGS | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | DIR | IND | REG | IMM | CY | AC | OV | P |
| **ADD A, byte** | A=A+byte | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **ADDC A, byte** | A=A+byte+C | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **SUBB A, byte** | A=A–byte–C | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **INC A** | A=A+1 | | | | | | | | ✓ |
| **INC byte** | byte=byte+1 | ✓ | ✓ | ✓ | | | | | |
| **INC DPTR** | DPTR=DPTR+1 | only DPTR | | | | | | | |
| **DEC A** | A=A–1 | only A | | | | | | | ✓ |
| **DEC byte** | byte=byte–1 | ✓ | ✓ | ✓ | | | | | |
| **MUL AB** | A=(B*A) % 256<br>B=(B*A) / 256 | only A and B | | | | 0 | | ✓ | ✓ |
| **DIV AB** | A=integer part of A/B<br>B=remainder of A/B | only A and B | | | | 0 | | ✓ | ✓ |
| **DA A** | Decimal Adjust | only A | | | | ✓ | | | ✓ |

### *2.3.5.2 Logic operations*

| MNEMONIC | OPERATION (C-style syntax) | ADDRESSING | | | | FLAG |
|---|---|---|---|---|---|---|
| | | DIR | IND | REG | IMM | P |
| **ANL A,byte** | A=A & byte | ✓ | ✓ | ✓ | ✓ | ✓ |
| **ANL byte,A** | byte=byte & A | ✓ | | | | |
| **ANL byte,#const** | byte=byte & const | ✓ | | | | |
| **ORL A,byte** | A=A \| byte | ✓ | ✓ | ✓ | ✓ | ✓ |
| **ORL byte,A** | byte=byte \| A | ✓ | | | | |
| **ORL byte,#const** | byte=byte \| const | ✓ | | | | |
| **XRL A,byte** | A=A ^ byte | ✓ | ✓ | ✓ | ✓ | ✓ |
| **XRL byte,A** | byte=byte ^ A | ✓ | | | | |
| **XRL byte,#const** | byte=byte ^ const | ✓ | | | | |

### *2.3.5.3 Accumulator manipulation*

| MNEMONIC | OPERATION (C-style syntax) | ADDRESSING | FLAGS | | | |
|---|---|---|---|---|---|---|
| | | | CY | AC | OV | P |
| **CRL A** | A = 0 | only A | | | | ✓ |
| **CPL A** | A = ~A | only A | | | | ✓ |
| **RL A** | Rotate A left by 1 bit<br>A = A << 1 | only A | | | | ✓ |
| **RLC A** | Rotate A left through Carry<br>A = (A << 1) + C<br>C = bit 7 of the original value of A | only A | ✓ | | | ✓ |
| **RR A** | Rotate A right by 1 bit<br>A = A >> 1 | only A | | | | ✓ |
| **RRC A** | Rotate A right through Carry<br>A = (A >> 1) + (C << 7)<br>C = bit 7 of the original value of A | only A | ✓ | | | ✓ |
| **SWAP A** | Swap nibbles of A | only A | | | | ✓ |

### *2.3.5.4 Bit-variable operations*

| MNEMONIC | OPERATION (C-style syntax) |
|---|---|
| **ANL C,bit** | C = C && bit |
| **ANL C,/bit** | C = C && !bit |
| **ORL C,bit** | C = C \|\| bit |
| **ORL C,/bit** | C = C \|\| !bit |
| **MOV C,bit** | C = bit |
| **MOV bit,C** | bit = C |
| **CLR C** | C = 0 |
| **CLR bit** | bit = 0 |
| **SETB C** | C = 1 |
| **SETB bit** | bit = 1 |
| **CPL C** | C = !C |
| **CPL bit** | bit = !bit |

### *2.3.5.5 Data move operations*

| MNEMONIC | OPERATION (C-style syntax) | ADDRESSING | | | |
|---|---|---|---|---|---|
| | | DIR | IND | REG | IMM |
| **MOV A,byte** | A = byte | ✓ | ✓ | ✓ | ✓ |
| **MOV byte,A** | byte = A | ✓ | ✓ | ✓ | |
| **MOV byte1, byte2** | byte1 = byte2 | ✓ | ✓ | ✓ | ✓ |
| **MOV DPTR,#const16** | DPTR = 16-bit immediate constant | | | | ✓ |
| **PUSH byte** | SP = SP+1<br>RAM[SP]= byte | ✓ | | | |
| **POP byte** | byte = RAM[SP]<br>SP = SP-1 | ✓ | | | |
| **XCH A,byte** | exchange the content of A and byte | ✓ | ✓ | ✓ | |
| **XCHD A,@Ri** | exchange low nibbles of A and RAM[Ri] | | ✓ | | |

### *2.3.5.6 External and code memory access*

| MNEMONIC | OPERATION (C-style syntax) |
|---|---|
| **MOVX A,@Ri** | A = XRAM[Ri] |
| **MOVX @Ri,A** | XRAM[Ri]= A |
| **MOVX A,@DPTR** | A = XRAM[DPTR] |
| **MOVX @DPTR,A** | XRAM[DPTR] = A |
| **MOVC A,@A+DPTR** | A = CODE[A+DPTR] |
| **MOVC A,@A+PC** | A = CODE[A+PC] |

#### *2.3.5.7 Jump and subroutine call*

| MNEMONIC | OPERATION (C-style syntax) |
|---|---|
| **JMP address** | Jump to address<br>PC = address |
| **JMP @A+DPTR** | Jump to A+DPTR<br>PC = A+DPTR |
| **ACALL address** | Call subroutine at 11-bit <address><br>PC = PC+2<br>SP = SP+1<br>RAM[SP] = PC lower order byte<br>SP = SP+1<br>RAM[SP] = PC higher order byte<br>PC = address |
| **LCALL address** | Call subroutine at 16-bit address<br>PC = PC+3<br>SP = SP+1<br>RAM[SP]= PC lower order byte<br>SP = SP+1<br>RAM[SP] = PC higher order byte<br>PC = address |

#### *2.3.5.8 Return from subroutines and interrupts*

| MNEMONIC | OPERATION (C-style syntax) |
|---|---|
| **RET** | Return from subroutine<br>PC = RAM[SP]*256 + RAM[SP-1]<br>SP = SP-2 |
| **RETI** | Return from interrupt<br>PC = RAM[SP]*256 + RAM[SP-1]<br>SP = SP-2<br>restore the interrupt logic to accept further interrupts |
| **NOP** | No operation |

#### *2.3.5.9 Conditional jumps*

Note that if a conditional jump occurs, the program counter is updated as PC=PC+address, where the address is an 8-bit two's complement number in the range of -128 to 127.

| MNEMONIC | OPERATION | ADDRESSING | | | |
|---|---|---|---|---|---|
| | | DIR | IND | REG | IMM |
| **JZ address** | Jump if A = 0 | only A | | | |
| **JNZ address** | Jump if A !=0 | only A | | | |
| **DJNZ byte, address** | Decrement and jump if not zero | ✓ | | ✓ | |
| **CJNE A,byte, address** | Jump if A != byte | ✓ | | | ✓ |
| **CJNE byte,#const, address** | Jump if byte != const | | ✓ | ✓ | |
| **JC address** | Jump if C = 1 | | | | |
| **JNC address** | Jump if C = 0 | | | | |
| **JB bit, address** | Jump if bit = 1 | | | | |
| **JNB bit, address** | Jump if bit = 0 | | | | |
| **JBC bit, address** | Jump if bit = 1; CLR bit | | | | |

### *2.3.6 Instruction timing and coding*

The CIP-51 architecture executes most of the operations in 1 or 2 system clock cycles. Depending on the specific device, the system clock can have maximum frequencies from 25 MHZ to 100 MHz; therefore, the fastest instruction execution time can be as low as 10 ns. The following table shows the distribution of the cycle time for the available instructions. Note that processors operating at clock frequencies above 25 MHz may use pipelining (prefetching instructions into a fast buffer) due to flash code memory access time limitations. This means that the processor may stall for a few clock cycles in some cases (for example, when a jump or a branching occurs).

| cycles | 1 | 2 | 2/4 | 3 | 3/5 | 4 | 5 | 4/6 | 6 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| **instructions** | 26 | 50 | 5 | 10 | 7 | 5 | 2 | 1 | 2 | 1 |

The CISC architecture of the 8051 processors allows instructions to be coded using 1, 2 or 3 bytes. The first byte is associated with the type of the instruction, while the remaining one or two identify the operands. A few examples are shown in the next table.

| instruction | 1. byte | 2. byte | 3. byte | cycles |
|---|---|---|---|---|
| **ADD A, Rn** | 0010 1nnn | | | 1 |
| **ADD A, #10** | 0010 0100 | 0000 1010 | | 2 |
| **ANL 15,#10** | 0101 0011 | 0000 1111 | 0000 1010 | 3 |
| **DIV AB** | 1000 0100 | | | 8 |

| | | | | |
|---|---|---|---|---|
| **JZ address** | 0110 0000 | relative address | | 2/4 |

### *2.3.7 Interrupt handler*

Event handling is one of the most important aspects of embedded programming. Events can be generated by peripherals such as timers, communication ports, and analogue-to-digital converters and also by changes of external signals. In the 8051 environment, events can generate interrupts, which can be serviced by subprograms. If an event occurs, a flag is set (which can even be polled by software) and an associated interrupt routine is called if enabled. The interrupt mechanism is visualised in Figure 2.4. When the event occurs, the system detects this within the system clock cycle time $\Delta t$ (the reciprocal of the system clock frequency). Upon completion of the currently running instruction (which can take from 1 to 8 cycles; see the previous chapter), an **LCALL** instruction is executed and the program jumps to the interrupt service routine. After processing, a **RETI** instruction is executed to return to the main program and restore the interrupt logic to accept further interrupts. One can easily see that the time elapsed from the event to the execution of the first instruction of the interrupt handler requires a minimum latency time and has some uncertainty as well.

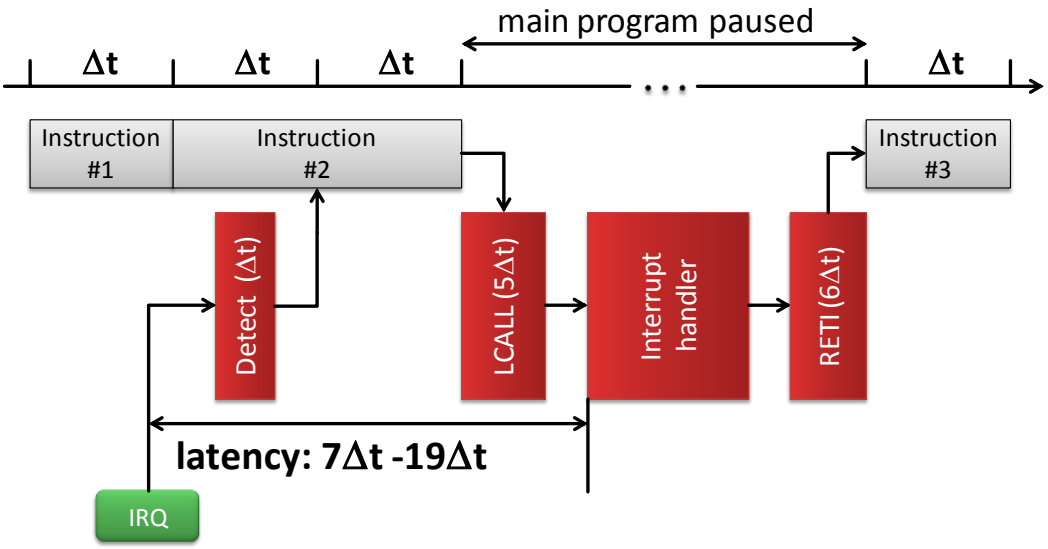


*Figure 2.4. Interrupt mechanism. The interrupt latency time varies from 7 to 19 system clock periods.*

It is very important to keep this in mind, since in a real-time application it can cause problems. For example, if a periodic interrupt is used to generate a square wave, this causes some fluctuation of the switching times, which should be considered, especially when switching times are short. For example, if a 100-kHz square wave is to be generated by a timer interrupt routine, the routine must be called 200000 times per second to change the signal state at every 5 µs. At a system clock frequency of 25 MHz, the clock period is 40 ns, so the latency time can vary from 7·40 ns to 19·40 ns, resulting in an uncertainty of (19-7)·40 ns=480 ns. This can cause a maximum error of 9.6% in the 5-µs switching time.

The main program can be interrupted at any time, even during a task requiring multiple instructions. This means that all temporary variables and register content modified by the

interrupt service routine must be saved at the beginning of the interrupt handling routine and must be restored upon return to the main program. Also note that the peripheral state or the input/output can also be changed during interrupt handling, which also needs careful attention.

If an interrupt service routine is running, another request can only be serviced if it has higher priority. Only two priority levels are provided, so no further interrupts can be serviced. The priority of the interrupts is defined by the bits of the **IP**, **EIP1** and **EIP2** registers. Correspondingly, there are only two priority levels: normal and high. If more interrupts are detected simultaneously, the higher priority interrupt will be serviced first. Since the interrupt flag set by the event can be cleared only when the associated interrupt routine is called, no interrupts are lost if multiple requests are detected or the request occurs during the servicing of another one. Of course, if a request is generated two or more times without servicing, only the last request can be serviced.

Interrupt sources are associated with a number that also defines priority (lower number means higher priority). The execution address of the interrupt routines is fixed and only 8 bytes are available up to the next address. Therefore, longer routines are located elsewhere and only a jump to that space is needed here.

A few interrupt flags are automatically cleared by the hardware when the service routine is called; all others must be cleared by the software – otherwise, the request will remain active and will be serviced continuously.

Interrupts can be individually enabled and disabled using the bits of the **IE**, **EIE1** and **EIE2** registers. **IE.7** (which can also be accessed as the SFR bit **EA**) is a global enable bit. Note that if an interrupt is enabled, it must have an interrupt handler code; otherwise, the processor can go into an uncertain state.

The interrupt sources available on C8051F410 processors are listed in the following table [6].

| Source | Execution Address | Number | Enable bit | Priority bit | Flag | |
|---|---|---|---|---|---|---|
| | | | | | name | Cleared by hardware |
| **Reset** | 0x0000 | - | | | - | yes |
| **/INT0** external | 0x0003 | 0 | IE.0 | IP.0 | IE0 | yes |
| **Timer 0** overflow | 0x000B | 1 | IE.1 | IP.1 | TF0 | yes |
| **/INT1** external | 0x0013 | 2 | IE.2 | IP.2 | IE1 | yes |
| **Timer 1** overflow | 0x001B | 3 | IE.3 | IP.3 | TF1 | no |
| **UART** | 0x0023 | 4 | IE.4 | IP.4 | RI, TI | no |
| **Timer 2** overflow | 0x002B | 5 | IE.5 | IP.5 | TF2H, TF2L | no |
| **SPI0** | 0x0033 | 6 | IE.6 | IP.6 | SPIF, WCOL,MODF, RXOVRN | no |
| **SMB0** | 0x003B | 7 | EIE1.0 | EIP1.0 | SI | no |
| **smaRTClock** | 0x0043 | 8 | EIE1.1 | EIP1.1 | ALRM, OSCFAIL | no |

| | | | | | | |
|---|---|---|---|---|---|---|
| **ADC0 Window Comparator** | 0x004B | 9 | EIE1.2 | EIP1.2 | AD0WINT | no |
| **ADC0 End of Conversion** | 0x0053 | 10 | EIE1.3 | EIP1.3 | AD0INT | no |
| **Programmable Counter Array** | 0x005B | 11 | EIE1.4 | EIP1.4 | CF, CCFn (up to six flags) | no |
| **Comparator 0** | 0x0063 | 12 | EIE1.5 | EIP1.5 | CP0FIF, CP0RIF | no |
| **Comparator 1** | 0x006B | 13 | EIE1.6 | EIP1.6 | CP1FIF, CP1RIF | no |
| **Timer 3 overflow** | 0x0073 | 14 | EIE1.7 | EIP1.7 | TF3H, TF3L | no |
| **Voltage regulator dropout** | 0x007B | 15 | EIE2.0 | EIP2.0 | - | no |
| **Port match** | 0x0083 | 16 | EIE2.1 | EIP2.1 | - | no |

# 3 Assembler and C programming

Programming 8051 microcontrollers requires special attention due to limited processing power, small memory space and the direct access of peripherals. No operating system is used in most cases; therefore, the programmer must take care of everything that the microcontroller does. The programmer must have extensive knowledge about the hardware, including memory types, instructions, SFRs, the interrupt handler and digital and analogue peripherals.

Simple programs can be written in assembler, but C is recommended for general-purpose code development. Although code optimisations are done by the C compiler, some fragments of code can be further enhanced by mixing assembler and C. C compilers allow inserting assembly code in C and C and assembly code can work on the same variables. C programmers can write efficient embedded code only if they know assembler as well.

## 3.1 SDCC C compiler

There are many 8051 C Compilers on the market. The most popular professional compiler is the KEIL C51 [3] and there exists an open-source alternative called Small Device C Compiler (SDCC) [4]. The free availability, good quality and the detailed documentation of SDCC make it an ideal tool to use in education. Here only the most important additions to C are mentioned that are needed to use the features of the 8051 processor.

Variables can be placed in different memory types; for this purpose, the compiler supports the declaration of storage classes:

```
__data unsigned char x;   // internal RAM
__xdata unsigned char x;  // external RAM
__idata unsigned char x;  // internal indirectly addressable RAM
__pdata unsigned char x;  // 8-bit addressed external RAM
__code unsigned char x=3; // constant in code memory
__bit b;                  // bit addressable RAM
__sfr __at 0x80 P0;       // SFR byte
__sbit __at 0xD7 CARRY;   // SFR bit
__xdata __at (0x4000) unsigned char x[16];   // external RAM, absolute address
__code __at (0x7f00) char Msg[] = "Message"; // code memory, absolute address
__bit __at (0x80) GPIO_0;                    // bit, absolute address
```

Inserting assembly into C can be done using the __asm and __endasm directives:

```
unsigned char x;
__asm              // beginning of assembly code fragment
    clr  a         /* C style comment */
    mov  R0,#0     // P0, C++ style comment
    mov  R1,#0x80  // C style hexadecimal constant
    mov  a,R2      // copy the content of R2 register to accumulator
    mov  _x,a      // accessing x declared in C
    jz   L1        // use of a label
    mov  R0,#0     // clear register R0
L1:
    mov  R1,#1     // load 1 into register R1
__endasm;          // end of assembly code fragment
```

The variable types are listed in the following table.

| type | width (bits) | default | signed range | unsigned range |
|---|---|---|---|---|
| **__bit** | 1 | unsigned | - | 0,1 |
| **char** | 8 | signed | -128–127 | 0–255 |
| **short** | 16 | signed | -32768–32767 | 0–65535 |
| **int** | 16 | signed | -32768–32767 | 0–65535 |
| **long** | 32 | signed | -2147483648<br>+2147483647 | 0–4294967296 |
| **float**<br>IEEE754 | 32 | signed | | $1.175494351 \cdot 10^{-38}$,<br>$3.402823466 \cdot 10^{+38}$ |
| **pointer** | 8-24 | generic | | |

Most of the variable types are the same as in standard C, but due to the limited resources, there are some exceptions. For example, the SDCC compiler allows defining bit variables using the __bit keyword. The variable can be placed in the bit addressable memory space, optimising memory usage. Floating-point arithmetic is supported; however, only single precision 4-byte wide float type variables can be used. This is fine in most embedded applications due to its 6-7 digits of precision. Double precision is not available, because it would take a long execution time and significantly longer code.

Generic pointers are rather special, since the 8051 microcontroller uses several different memory types. The 3-byte wide generic pointer defines the address in two bytes and the memory type (internal RAM, external RAM or code memory) on the third byte. Of course, the programmer can declare a pointer that points explicitly to an internal memory location. This pointer is stored in a single byte since only 256 different locations are possible.

Microcontroller programming often requires the manipulation of bits. Here are two simple examples:

```
x = x & ~(1 << 3);  // clearing a bit
x = x | (1 << 3);   // setting a bit
```

Working with integer numbers that are not 8, 16 or 32 bits long is also common. Left or right shifting may be required, but care must be taken concerning signed and unsigned numbers, since the behaviour of the shift operator is different for signed and unsigned numbers. For example, to handle a 2s complement 12-bit number (the four most significant bits are in ADCH and the eight least significant bits are in ADCL), one may use the following code:

```
short x;  // define a signed 16-bit integer variable
x = (ADCH << 12) + (ADCL << 4);                // left justified
x = ((signed short)(ADCH << 12) >> 4) + ADCL;  // right justified
```

In most cases, unsigned integers are used for the data of the peripherals (such as counter value or ADC value). The programmer should always declare the variable as unsigned if it contains an unsigned number. However, the use of negative constants can help in some cases, especially when calculating the value used in timer programming (see Chapter 5):

```
unsigned short x;  // define an unsigned 16-bit integer variable
x = -100;  // this is equivalent to 65536-100, i.e. 65436
```

Note that 65536 cannot be represented by an unsigned short variable and long arithmetic would take more time and longer code.

## *3.2 Interrupt programming in assembler*

A simple assembler interrupt handler example code is listed below. At the beginning, the registers in use are pushed onto the stack and restored at the end of the routine. The interrupt pending flag (in this example **RI**) is cleared. Note the use of assembler-style comments.

```
push ACC    ; ACC (SFR access of A) to the stack
push PSW    ; status register to the stack
clr  RI     ; clear interrupt flag
mov  A,SBUF ; A is changed here
add  A, #1  ; A and PSW are changed here
mov  P0,A   ; copy the content of the accumulator to port P0
pop  PSW    ; PSW restored here
            ; reverse order!
pop  ACC    ; ACC (A) is restored here
reti        ; return to the main program
```

If the **R** registers are used, they must be saved and then restored as well. However, the 8-byte register bank can be moved to four memory locations; therefore, the interrupt routine can use one bank while the main code uses another bank.

```
push PSW    ; status register to the stack
mov  PSW,#8 ; use register bank #1
; use R registers here
pop  PSW    ; PSW and the register bank selection is restored here
```

The following complete assembler code illustrates the use of a timer interrupt to make an LED blink. The system clock after reset for the C8051F410 processor is 191406 Hz, and its 16-bit Timer 2 runs with 1/12 of this rate by default: 191406/12 Hz ≈ 15950 Hz. Since the interrupt occurs when the 16-bit timer overflows, 15950 steps are needed to reach $2^{16}$=65536 in order to wait 1 second before overflow. Therefore, the initial value of the timer should be set to 65536-15950 = 49586 = 0xC1B2, and this value will be reloaded upon overflow automatically. This way, a periodic interrupt will be generated every second. Note that the detailed description of the peripherals can be found in the following chapters.

```
$include (C8051F410.INC) ; load the definitions used for the C8051F410 MCU

LED     EQU  P0.2        ; the LED is connected to bit 2 of port 0.

CSEG at 0000h
   jmp  Main             ; reset, jump to the label 'Main'

ORG 002Bh                ; Timer 2 interrupt location
   anl TMR2CN,#07Fh      ; clear interrupt flag
   cpl LED               ; complement LED
   reti                  ; return from interrupt
Main:
   anl  PCA0MD,  #0BFh   ; switch watchdog off
   mov  PCA0MD,  #000h   ; switch watchdog off
   mov  XBR1,    #040h   ; enable the crossbar to allow input and output
   mov  TMR2RLL, #0B2h   ; set the Timer 2 reload register (low and high bytes)
   mov  TMR2RLH, #0C1h   ; to provide 1-Hz interrupt rate
```

```
    mov  TMR2L,   #0B2h  ; Timer 2 counter initial value
    mov  TMR2H,   #0C1h  ; is the same as the reload value
    mov  TMR2CN,  #004h  ; Start Timer 2 now
    mov  IE,      #0A0h  ; enable global interrupts and Timer 2 interrupt
    jmp  $               ; repeat forever, interrupt routine will blink the LED
END
```

## 3.3 *Interrupt handling in C*

The SDCC C compiler for the 8051 family of processors supports interrupt programming. If a function is intended to be an interrupt handler, it must be declared accordingly. The programmer should use the __interrupt keyword and include the number of the interrupt to identify which interrupt will be handled. For example:

```
void InterruptHandler(void) __interrupt 5
```

or using predefined constants

```
void InterruptHandler(void) __interrupt INT_TIMER2
```

This code defines an interrupt handler (no return value or input arguments can be defined) for the Timer 2 interrupt that is numbered as 5. The location of the **R0**—**R7** register bank can also be defined:

```
void InterruptHandler(void) __interrupt 5 __using 1
```

where the number following __using keyword defines which one of the four possible register banks are used (the default is 0). This can help the compiler to generate faster code since saving/restoring the registers is not necessarily needed.

An interrupt handler can use local variables, which are initialised in any execution of the routine. A simple example is the use of temporary variables. However, in some cases a variable must retain its value after exiting from the interrupt handler routine. For example, if the code must count how many interrupts are generated, a counter value must be incremented each time the interrupt routine is called. This variable can be declared as a global variable just at the beginning of the code, but if it is used in the interrupt routine only, it is best to hide the variable from other parts of the code. In this case, the variable should be declared in the interrupt handler routine using the static keyword. The following example code toggles the state of an LED upon every hundredth Timer 2 overflow interrupt request. The static variable named ‘counter’ counts how many requests are detected, and if this number reaches 100, the LED is toggled. Since the counter value is only used in the interrupt routine, it can be declared within the scope of the routine. Note that the initialisation of the variable is done only when the program starts.

```
/***********************************************************************
Timer 2 interrupt handler
***********************************************************************/
void IntHandler(void) __interrupt 5
{
    static unsigned int counter = 0; // will be initialised only once!
    TMR2CN&=~0x80;  // or TF2H=0, clear interrupt pending flag
    counter++;      // increment the value of the counter variable
    if (counter == 100) // the routine has been called 100 times
    {
        counter = 0;    // reset counter
        LED = !LED;     // complement LED
    }
}
```

If a variable is used both in an interrupt routine and in other parts of the program then *it must be declared as* volatile:

```
// define the global variable that is used both in the interrupt routine
// and in the main program
volatile unsigned char counter;

/***********************************************************************
Timer 2 interrupt handler
***********************************************************************/
void IntHandler(void) __interrupt 5
{
    TMR2CN&=~0x80;  // or TF2H=0, clear interrupt pending flag
    counter++;      // increment the value of the counter variable
}
```

The volatile keyword tells the compiler that the variable can be changed at any time, so it cannot be assumed to remain unchanged in a sequence of a few lines of computations. It is a typical error to get unexpected results due to missing volatile declarations.

There are several considerations to be kept in mind if both interrupts and regular code work on the same data. The main program may work on data in several assembler instructions that are hidden from the programmer. For example, checking the value of a 16-bit number needs several assembler instructions: the higher- and lower-order bytes must be separately checked – it is not an *atomic* operation. If an interrupt routine changes the value of this integer during this process, an unexpected error can occur. Therefore, non-atomic operations must be protected.

One solution is the use of critical blocks as shown below:

```
volatile unsigned short x;  // variable declaration
.
.
/***********************************************************************
Interrupt handler routine
***********************************************************************/
void IntHandler(void) __interrupt 5
{
    TF2H=0;  // clear interrupt pending flag
    x++;     // increment the value of x
}
.
.
.
__critical  // define the critical block
{
    if (x>1024) DoSomething();  // here x cannot be changed by IntHandler()
}
```

At the beginning of the critical block, the compiler disables interrupts (saves then clears **EA**) and at the end re-enables them if necessary (restores the value of **EA**); therefore, no interrupt can be executed within the critical block. Note that this may cause extra interrupt latency and even missing interrupts if the critical block needs too much time to complete.

It is also possible to protect a variable from being modified by the interrupt routine by introducing a user flag as illustrated in the following example.

In the main code:

```
volatile __bit protect_x;   // flag variable
volatile unsigned short x;  // this variable can be changed in the
                            // interrupt handler routine

protect_x=1;                // switch protection on
if (x>10) DoSomething();    // here x cannot be changed
protect_x=0;                // switch protection off
```

In the interrupt handler:

```
if (!protect_x)             // allow changes only if protect_x is 0
{
    x=(ADC0H << 8) | ADC0L; // do the change of x
}
```

Note that some events generate the same interrupt. For example, interrupt 6 corresponds both to serial port receive and transmit. Therefore, the interrupt handler must check if the **RI** or the **TI** interrupt flag is set and execute the code accordingly.

## *3.4 Interrupt programming guidelines*

Interrupt programming is rather difficult, as there are many potential pitfalls. Response time, latency, processing time, variable and memory content, priority, peripheral status, simultaneous requests and a lot more are all to be considered carefully. Debugging is not easy due to the complexity and the differences between real-time versus single stepping operating modes. Here are some guidelines to follow to reduce the probability of unexpected behaviour.

- If the interrupt handler and the rest of the code work on the same data, synchronisation must be carefully designed.
- Atomic and non-atomic operations should be identified. Temporary results of non-atomic operations on data must be protected from being modified by an interrupt service routine.
- Before enabling the interrupt, the corresponding peripheral must be configured and the variables used must be initialised.
- An enabled interrupt must have an interrupt handler routine.
- The stack size must be set to have enough space for saving and restoring variables and for subprogram calls. It is best to set the initial value of the stack pointer (**SP**) to the first free location of data memory, just above the variables. In this case, all free memory is available as stack.
- Interrupt routines should take as short a time as possible, and only the most important processing that cannot be done by the main program should be performed here.
- Too frequent interrupt calls can slow the processor down; too frequent multiple concurrent interrupt requests can cause a failure to service certain requests.
- Multiple interrupts can generate extra interrupt latency time – another reason to keep interrupt service routine execution time as short as possible.
- Interrupt pending flags must be cleared in the interrupt service routine.
- Interrupt priorities must be taken into account. Priority of critical interrupts requiring fast response must be set to high.
- Consider using different register banks for interrupts.
- Several interrupt flags can be associated with the same interrupt routine; therefore, the routine must handle all of them.

- Do not mix event handling by polling the interrupt pending flag with event handling by an interrupt service routine. Choose between the two possibilities.
- Do not use library routines in interrupt routines. They can be slow and can be non-reentrant. Only reentrant functions can be called while one instance is already running. On the other hand, reentrant functions are slower and need more resources. For example, floating-point arithmetic and 16-bit and 32-bit integer multiplication, division and modulus operations use non-reentrant support functions. See the SDCC manual how to overcome this limitation.

## *3.5 Using an integrated development environment and the associated tools*

Silicon Laboratories provides a free integrated development environment (IDE) and several other software tools to support code development [2].

Many different compilers can be integrated with the IDE, including the open-source and free SDCC C compiler. In the *Tool Chain Integration* menu item the compiler can be selected.

Projects can be created and header and C source files or libraries can be added to the project as usual in IDEs.

The IDE handles the USB debug adapter that connects the PC to the target microcontroller. The adapter allows the downloading of the compiled code and also provides debug functions. After compilation, the code can be downloaded.

Breakpoints can be defined, so after starting the code, real-time execution will automatically stop when a breakpoint is found. This means that the real system can be monitored and no simulation is performed. During debugging both the assembly and C code can be viewed.

Another very useful feature is the watch window, which can show the actual content of several variables. Besides this, there are many debug windows available to view and even change the contents of the registers, memories and peripherals of the 8051.

Single stepping, full speed execution, run-to-cursor execution are all possible. An example screenshot of the IDE can be seen in Figure 3.1. Solid red circles indicate breakpoints, while the blue bar shows the current source line being executed. Keyword highlighting is also provided.

On the right the peripheral watch windows – programmable counter array (PCA) window; the 8051 register (including the program counter, **PC** and accumulator, **ACC**) window; the disassembly windows and the variable watch window (which shows the **INT0counter** and the **PCAcounter**) – can be seen. Red colour indicates recently changed values.

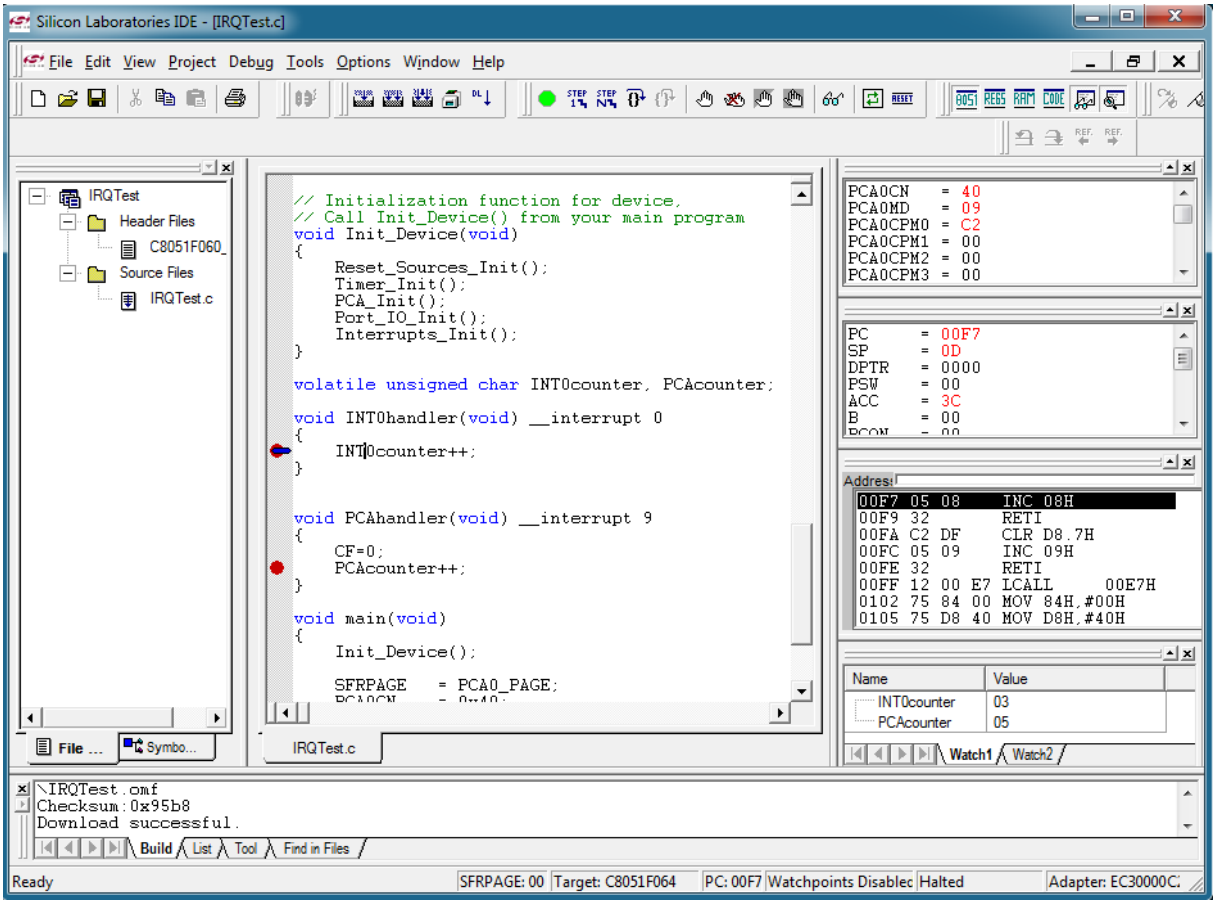


*Figure 3.1. Debugging in the Silicon Laboratories IDE.*

Note that peripherals are stopped if the program is paused in the debugger, and after a single-step operation the code is halted again. This means that all peripherals are stopped at this point. This must be kept in mind, because full-speed execution might differ significantly. Examples are given below.

- Assume that the voltage reference is switched on in a program line and in the next line an analogue-to-digital conversion is initiated. Since the voltage reference needs a settling time of a few milliseconds, at full-speed execution the A/D conversion value will be invalid (switching the reference on needs only a few clock cycles), while in single stepping mode there is enough time for the voltage reference to settle.
- Sending a byte over a serial port is initiated by writing the **SBUF** register, for example: **SBUF**=0xAA;. Since the individual bits of SBUF are transferred at a certain rate (at each overflow of a timer), several clock cycles are needed to complete the transfer. Therefore, if the user places a breakpoint after **SBUF** loading (that is a two-cycle instruction) or performs single stepping, the data will not be transferred, because the timers will be halted.

## *3.6 Config Wizard*

The C8051Fxxx processors have a very rich set of peripherals that are configured with many SFRs — typically each has independent configuration bits. Therefore, it would be very hard to read the datasheet and set the individual bits of these SFRs accordingly, and the probability of making an error would be rather high. The Config Wizard 2 free graphical user interface development tool helps to configure the processor and its peripherals very efficiently. After

choosing the processor, it is possible to configure any of its peripherals by dialogue boxes and the corresponding source code will be generated in C or assembly format. This code can be copied into the user code. Figures 3.2 and 3.3 show two examples: the Port input/output and the Timer configuration dialogue boxes.

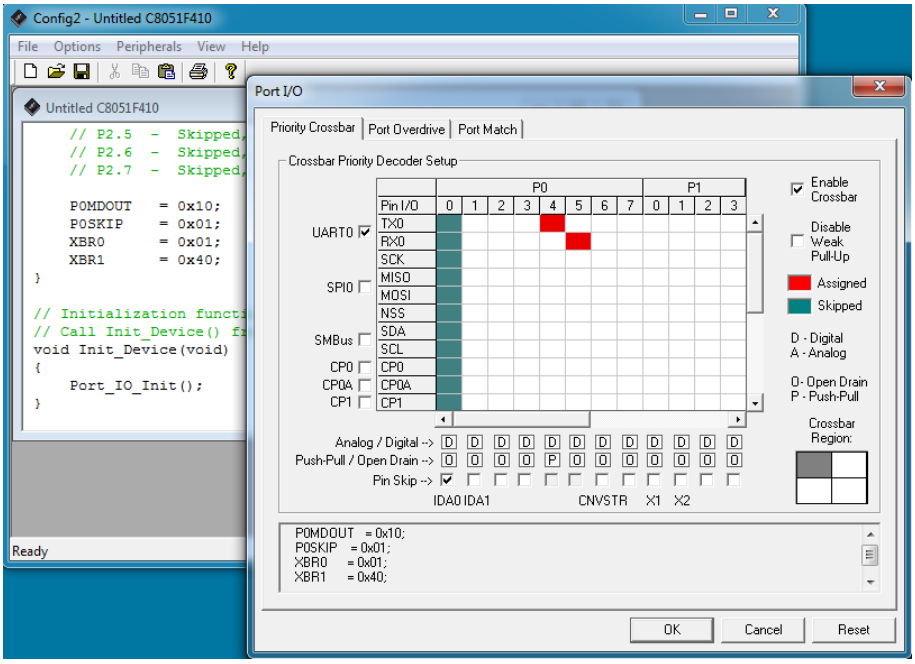


*Figure 3.2. Port input/output configuration dialogue box.*

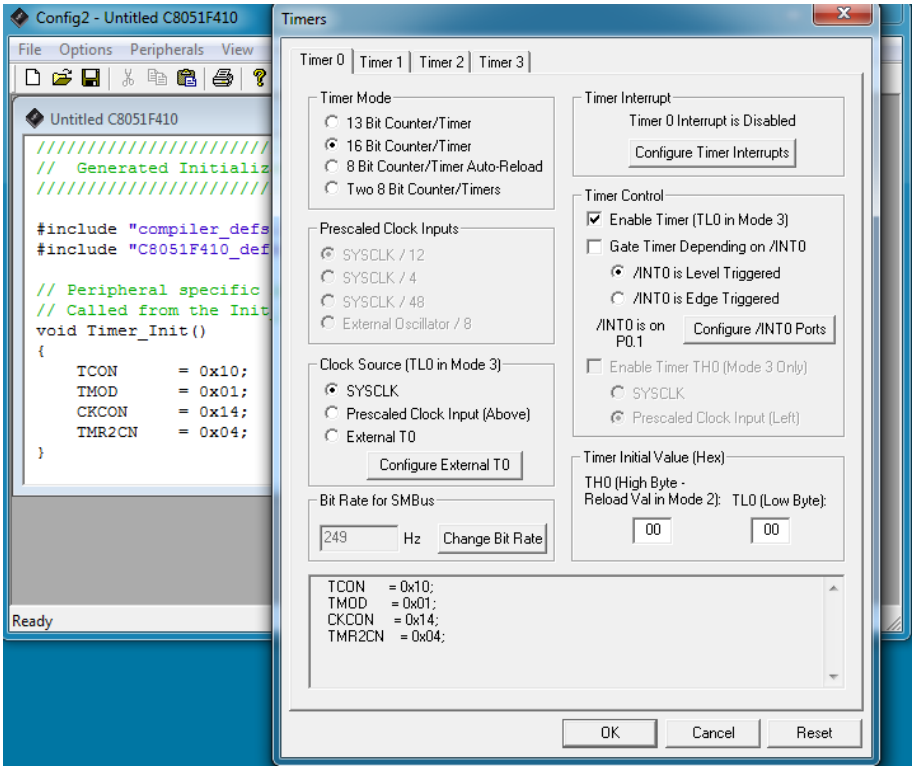


*Figure 3.3. Timer configuration dialogue box.*

# 4 Digital input and output; crossbar

Microcontrollers provide ports for general-purpose digital input and output signals [6]. The ports are organised in 8-bit groups (named **P0**, **P1**, etc.), but bits can also be accessed individually (for example, **P1.3** in assembler or **P1_3** in SDCC to access the third bit of port **P1**). All port bits are associated with the pins of the package of the chip and can be configured as input or as output, and have several operating modes. Some port pins can also be configured in analogue mode.

## 4.1 The I/O structure

The port structure can be seen in Figure 4.1.

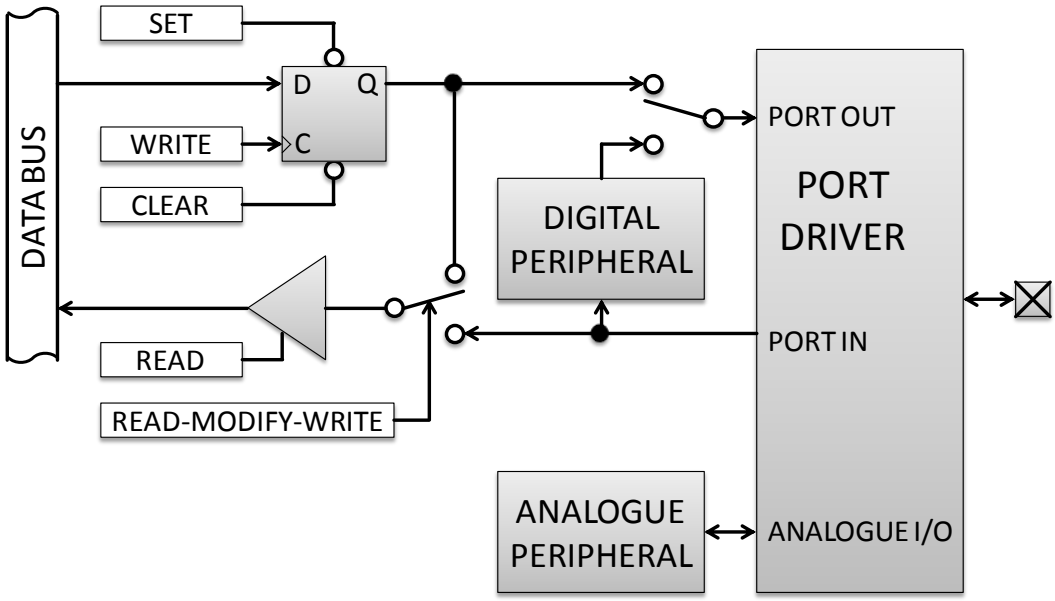


*Figure 4.1. I/O port structure.*

Writing to a port (for example, **MOV P0, #1** in assembler or **P0=1** in SDCC) means writing the data into a D-latch that is connected to a port pad via the port driver. During reading from a port, the port pad is connected to the internal data bus. Note that read-modify-write instructions (for example, **INC P1**, **ANL P1,#1** in assembler and **P1++, P1&=1** in C) do not read the state of the external signal itself, but rather use the output of the D-latch instead to guarantee consistent operation.

The simplified schematic of the port driver is shown in Figure 4.2. A complementary transistor pair can pull the line down to GND or up to Vdd (power supply of the driver stage) and a third transistor can switch on a weak pull-up. The port input uses a Schmitt trigger to guarantee valid logic levels for slowly changing or noisy signals. In order to use the analogue mode, all transistors must be switched off and the input Schmitt trigger must also be disabled.

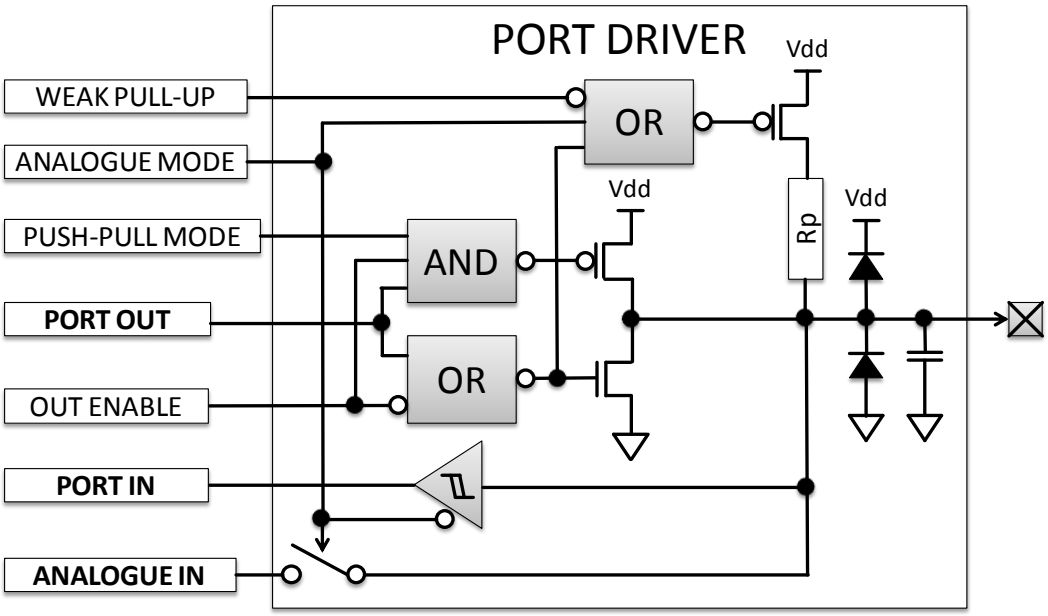


*Figure 4.2. Simplified schematic of the I/O port driver. Bold indicates internal I/O signals.*

### *4.1.1 Port input*

The port can be configured as digital input by switching off the output drivers. Therefore, it is important to *write logic 1 to the corresponding port bit,* otherwise the transistor connected to the ground will short-circuit the port pin to the ground. Push-pull mode must be disabled. The weak pull-up ($R_p$, roughly 100 kΩ, actually a weak P-channel FET) can be enabled or disabled globally for all port pins. When disabled, the leakage current is typically 10 nA at room temperature and is guaranteed to be less than 1 μA. This must be considered in analogue mode, whereas digital signals will not typically be affected by this small current that is matched with the specifications of other CMOS devices. The input capacitance is close to 5 pF and the diodes protect the internal circuitry against electrostatic discharge (ESD). The simplified equivalent schematic is shown in Figure 4.3.

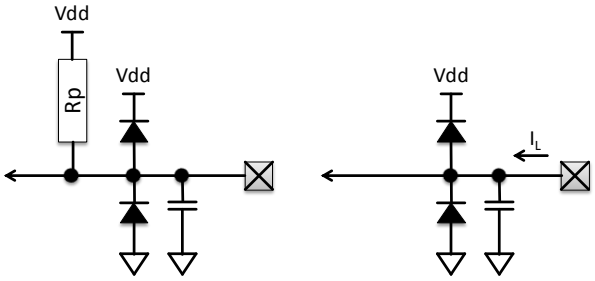


*Figure 4.3. Port input configurations. On the left, the digital input with weak pull-up is shown. On the right, the digital input with no pull-up and the analogue input can be seen. The typical $I_L$ leakage current is about 10 nA but can be as high as 1 μA.*

Note that the diodes protect the inputs from electrostatic discharge (ESD) and from over- or undervoltage, but the current cannot exceed the specifications given in the absolute

maximum ratings section in the datasheet. Since the supply voltage is less than 5 V, some ports provide 5 V tolerant inputs. In this case, the diode connected to Vdd is missing.

### *4.1.2 Port output*

The ports can operate either as open-drain or as push-pull outputs.

In open-drain mode, the weak pull-up can be switched on (reset default) or off. Logic 1 state can only be set by the large value (roughly 100 kΩ) pull-up resistor; therefore, the port cannot be loaded and signal transition from 0 to 1 will be rather slow, since the external capacitances can only be charged through the resistor. For example, a loading of 50 pF will reduce the rise time (90% of the final value) to about 10 μs. External pull-up resistors (down to about 1 kΩ) can make the switching faster and can source more current to the load at the expense of a larger quiescent current when 0 is written to the port bit. Some peripherals (such as I²C) require open drain mode.

In push-pull mode the output drive strength is symmetric, and the port can sink and source large currents and guarantee fast switching from 0-to-1 and from 1-to-0. Therefore, *it is strongly recommended to use push-pull mode for the output in most applications*, especially for communication peripherals, in order to avoid data corruption.

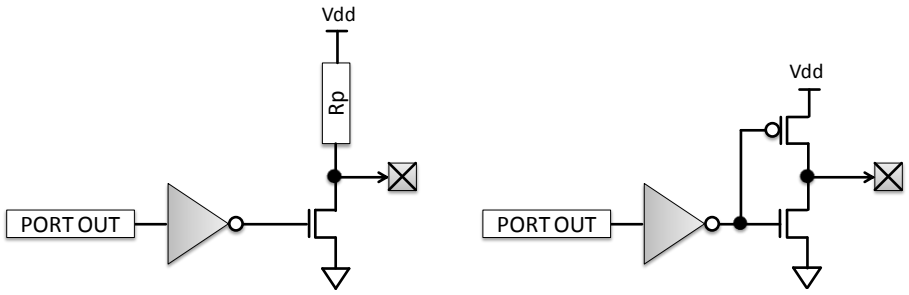


*Figure 4.4. Open-drain and push-pull output modes.*

## *4.2 Crossbar*

After reset, the ports are not connected to the core and all peripherals are idle. Port pins can be associated with the port latches or with the enabled peripherals, which can output or input signals (see Figure 4.5). The priority crossbar provides a flexible way to connect the internal peripherals and port latches to the port pins. If it is enabled, the port pins are accessible. If a peripheral is used, its signals are associated with port pins. Peripherals are numbered and the port pins are associated in this order with the enabled peripherals. For example, if peripheral #1 is enabled with two signals and peripheral #5 is enabled with three signals, peripheral #1 will be connected to the first two pins (**P0.0** and **P0.1**), while peripheral #5 will be associated with the next three pins (**P0.2**, **P0.3** and **P0.4**). The state of these pins cannot be modified by writing to the port latches but their state can be monitored by reading the corresponding port bit. The push-pull or open-drain settings can still be set by firmware.

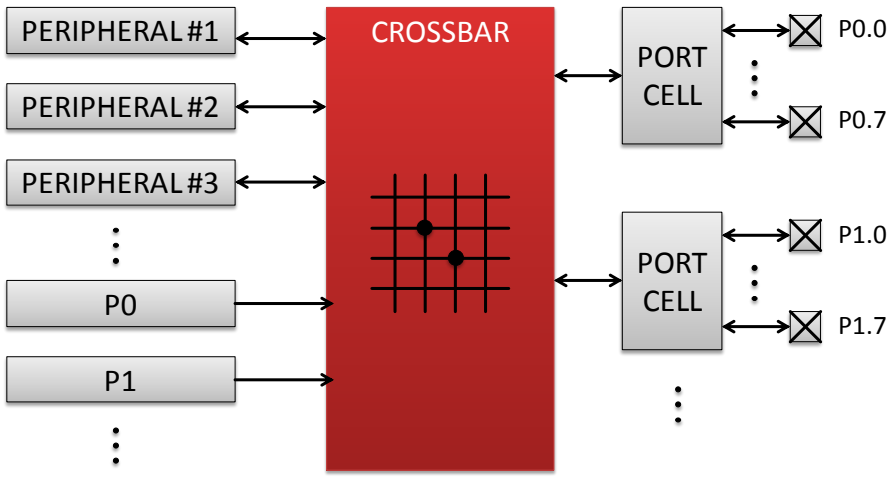


*Figure 4.5. The crossbar assigns peripherals and port latches to port pins.*

Crossbar settings and peripheral configuration are typically done just after reset. However, it is possible to reconfigure the system during execution. In this case, the crossbar must be disabled first, then can the changes be made before re-enabling the crossbar. Note that during this process the port pins may exhibit transitions, which must be tolerated by the system.

## 4.3 Port I/O applications

In this chapter port input and output application examples will be shown. The port I/O allows realising various user interfaces and communication with external circuits.

### 4.3.1 Reading buttons and switches

One of the simplest and most common digital input types is the state of a button or a switch. Figure 4.6 shows four ways of connecting buttons to the port pins. The most popular connection uses a pull-up resistor and a grounded button, as can be seen in the two configurations on the left. If the button is pressed, the corresponding logic value is 0. A capacitor is sometimes used to eliminate bouncing and to reduce noise. Positive logic can be realised by swapping the resistor and the button: in this case logic 1 is obtained when the button is pressed.

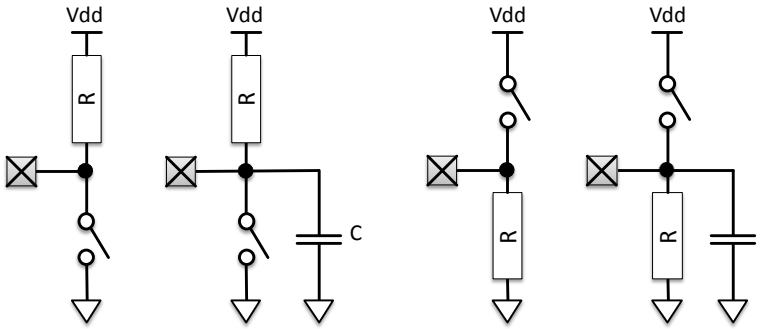


*Figure 4.6. Four ways of connecting a button or switch to the port pins. The two configurations on the left represent negative logic, while the other two correspond to positive logic. The value of R is typically 10 kΩ.*

Figure 4.7 shows that if the internal weak pull-ups ($R_p$) are switched on, the external pull-up resistor can be eliminated. Although the external capacitor may cause voltage at the input to change slowly, the internal Schmitt trigger ensures reliable operation.

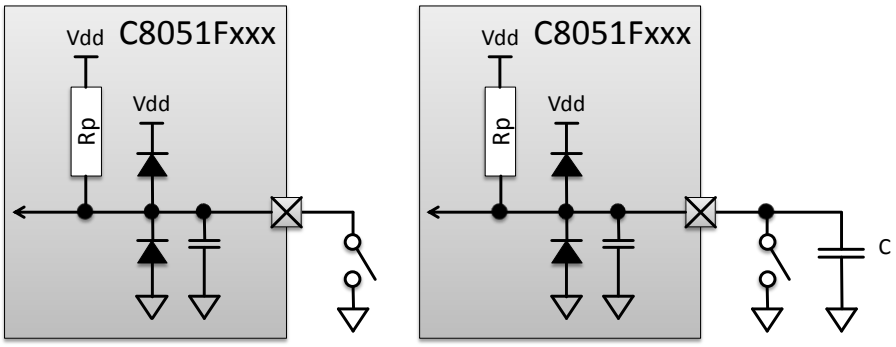


*Figure 4.7.*

The following, very simple source code shows an example of reading the state of a button connected to the third bit of port **P0**:

```
#define BUTTON_ON (!P0_3) // define an alias to access
                          // the port bit of the button

/*************************************************************************
The main function
*************************************************************************/
void main(void)
{
    while (1)  // infinite loop; the microcontroller never stops
    {
        if (BUTTON_ON)          // if the button is pressed
        {
            while (BUTTON_ON);  // wait while button is released
            DoShortProcess();   // some process to be completed
        }
    }
}
```

There are many problems associated with the code above. For example, potential bouncing is not handled and during process execution button pressings are lost.

An improved code uses a timer interrupt to detect button pressing only if the button is pressed for a period of at least 100 ms:

```
#define BUTTON_ON (!P0_3) // define an alias to access
                          // the port bit of the button
#define BUTTON_ON_TICK 10 // number of ticks to be counted
                          // defines the minimum time for button detection

volatile bit ButtonPressed; // variable to indicate if the button
                            // has been pressed

/*************************************************************************
Timer interrupt handler routine
*************************************************************************/
void TMRHandler __interrupt TMRVECTOR // 10-ms period
{
    // static variables retain their values upon exiting the function
```

```
    static bit buttonstate=0; // this bit stores the state of the button
    static bit detected=0;    // set if button pressing is detected
    static unsigned char counter=0; // counter for ticks

    if (ButtonPressed)         // button pressing not yet handled
        return;                // nothing to do
    if (BUTTON_ON)             // button is in a pressed state
    {
        if (buttonstate)       // it has already been pressed
        {
            counter++;         // increment time interval counter
            if (counter == BUTTON_ON_TICK) // if enough time has elapsed
                detected=1;                // pressing detected
        }
        buttonstate=1;       // save button state
    }
    else                     // it is in a released state
    {
        buttonstate=0;       // save button state for the next function call
        counter=0;           // reset time interval counter
        if (detected)
        {
            ButtonPressed=1; // notify main program
            detected=0;      // reset; end of detection; enable next detection
        }
    }
}
// in the main function:
   …
    if (ButtonPressed)    // button pressing has been detected
    {
        DoSomething();    // execute a process
        ButtonPressed=0;  // clear flag to enable further detections
    }
```

### *4.3.2 Reading a keyboard*

A more advanced user input interface is the keyboard. Keys are arranged in columns and rows. Columns and rows have associated wires, which are connected to each other if a key is pressed. Figure 4.8 shows how to interface the keyboard to the microcontroller. The wires of the rows are connected to port pins configured as inputs (**Pn.3** to **Pn.6**), while the columns are driven by port pins configured as outputs (**Pn.0** to **Pn.2**). Note that the optional pull-up resistors may be used on the inputs (**Pn.3** to **Pn.6**).

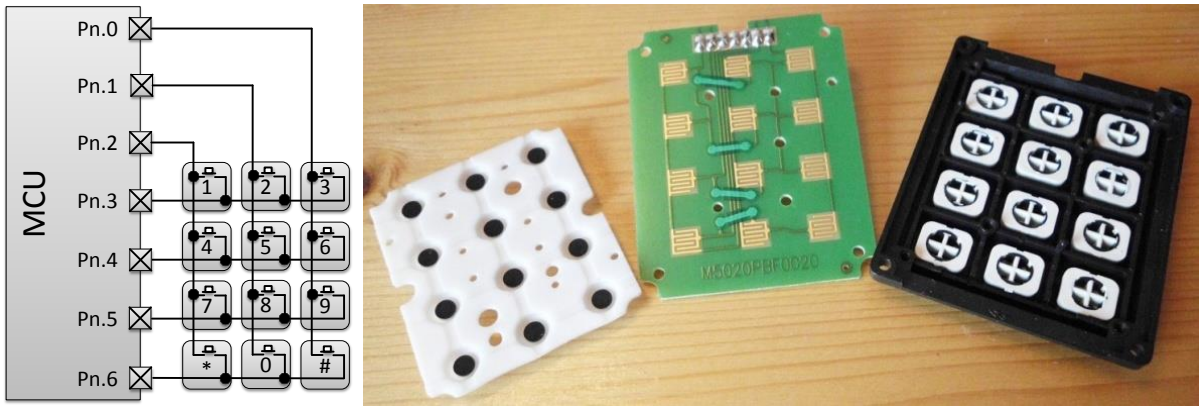


*Figure 4.8. Connecting a keyboard to the microcontroller port.*

The microcontroller typically scans all keys to determine which key is pressed. To do so, one of the column wires must be pulled down (by clearing one of the **Pn.0**, **Pn.1** or **Pn.2** outputs while the others are at logic 1) and check which row wire is at logic low. This procedure must be performed on all columns to determine which keys are pressed. In most cases, the algorithm can be stopped if a key if found to be pressed and no further keys need to be checked.

### *4.3.3 Driving LEDs*

LEDs are the simplest indicators that can inform the user about logic values. They can be connected in negative or positive logic, i.e., they can be lit by writing either logic low or logic high to the corresponding port bit. Figure 4.9 shows all connections. Open-drain output mode can only be used if the anode is connected to the supply, while push-pull mode can be used in both connections.

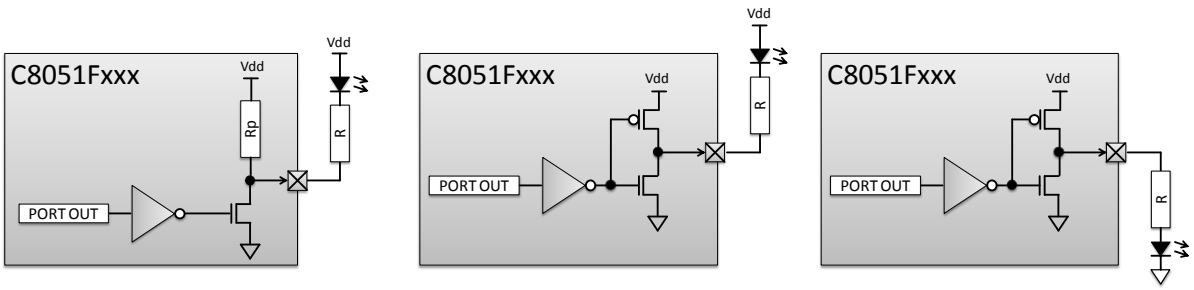


*Figure 4.9. An LED can be connected between a port pin and the supply or ground via a series resistor that sets the current. The push-pull configuration is needed to drive an LED whose cathode is grounded. Note that the push-pull mode can be used in both cases.*

The current setting resistor should be selected to provide enough light intensity, but keep in mind that output current of the port is limited and if many LEDs are driven, the total current sourced or sunk can be too large. Values from 330 Ω to 1 kΩ are typical. External drivers or transistors can be used to overcome this limitation.

### 4.3.4 Driving 7-segment displays

The 7-segment display contains 7 LEDs to display a decimal digit and one LED to represent an optional decimal point if multiple displays are used. The anodes or cathodes of the LEDs are connected to support positive or negative logic. Figure 4.10 shows the common-anode version associated with the negative-logic mode, which allows the port output to be configured either in open drain or in push-pull mode.

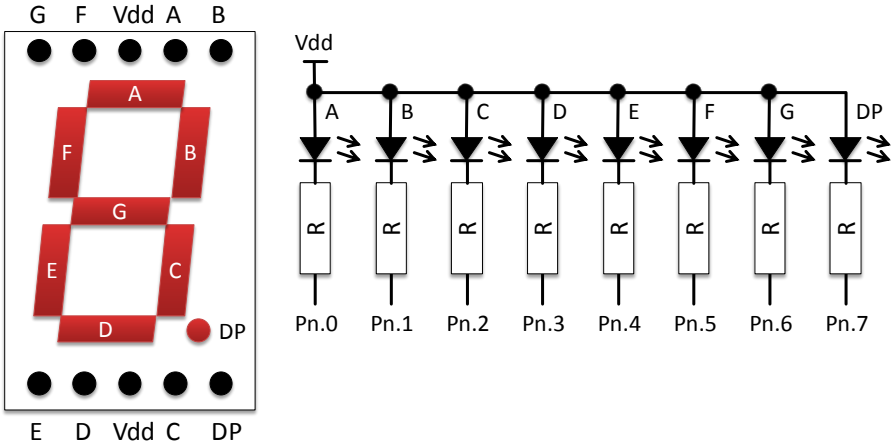


*Figure 4.10. An 8-bit port can drive a 7-segment display.*

The following table shows the port bits and the port byte to be written to display a specific digit.

| Digit | G | F | E | D | C | B | A | PORT |
|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0x3F |
| **1** | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0x06 |
| **2** | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0x5B |
| **3** | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0x4F |
| **4** | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0x66 |
| **5** | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0x6E |
| **6** | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0x7E |
| **7** | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0x07 |
| **8** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0x7F |
| **9** | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0x6F |

Note that multiple 7-segment displays can be connected to the same port provided that only one is enabled at a time. For example, the extension board (see the Appendix) has two 7-segment displays connected to Port 2 and one bit (**P1.3**) selects the display whose common cathode will be connected to Vdd. Therefore, only one display can be active at a time, but if they are toggled quickly enough, the user will see both displays working with different numbers. Of course, the brightness will be halved in this case.

### 4.3.5 Driving alphanumeric LCD displays

A much more powerful popular user interface is the alphanumeric liquid crystal display (LCD). The microcontroller can communicate with its integrated processor over a special 8-

bit parallel interface that can also be configured as a 4-bit interface. A detailed description can be found in the datasheet of the HD44780 or a compatible processor [16].

The LCD display can communicate with the processor over its parallel interface. The 8-bit bidirectional bus can be connected to one port. This port can be configured in open-drain mode, but in this case external pull-up resistors may be needed (3 kΩ-10 kΩ) to ensure the short rise time of the signals. Alternatively, the port can be configured in push-pull mode and should be changed to open drain only for read operations. The 3 control lines are driven by the port bits of the microcontroller; push-pull mode is strongly recommended. The R/W line selects between read (R/W=1) and write (R/W=0) operations. RS selects which one of the two register sets are written or read. If RS is logic high, then the display RAM is accessed and characters can be written to the display; otherwise instructions to the LCD display can be sent (for example clearing the display). A pulse on E reads or writes the data. Vdd (5 V in most cases) and GND are the supply lines. The voltage input V0 is used to set the contrast of the display. If the LCD display has internal backlighting LEDs, pins 15 and 16 can be used to power these. The anode and cathode can be connected in both ways; the datasheet must be consulted to determine the proper connection. Figure 4.11 shows the connector pinout of the LCD display.

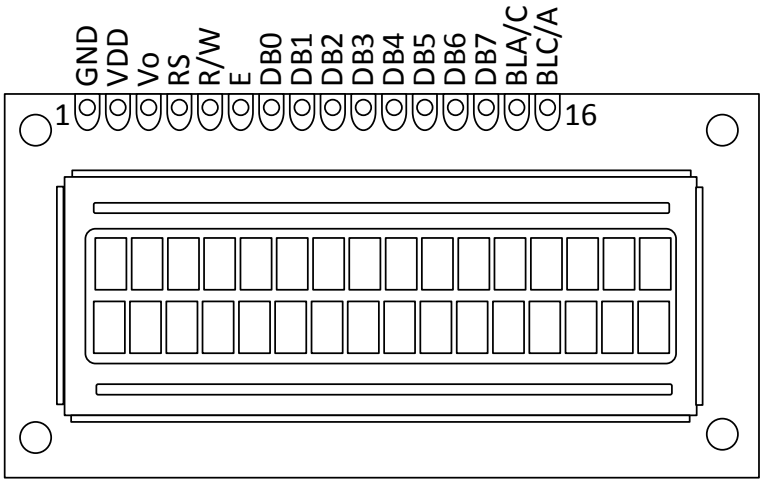


*Figure 4.11. Pinout of the standard LCD display connector.*

| Instruction | data | | | | | | | | comment |
|---|---|---|---|---|---|---|---|---|---|
| | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | |
| **Clear display** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Clears the whole display |
| **Return home** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | - | Reset cursor and entry mode |
| **Entry mode set** | 0 | 0 | 0 | 0 | 0 | 1 | ID | S | Entry mode:<br>ID=1:cursor right<br>S=1:entire display shift |
| **Display on/off** | 0 | 0 | 0 | 0 | 1 | D | C | B | D=1 display on,<br>C=1 cursor on, B=1 cursor blinks |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Cursor or display shift** | 0 | 0 | 0 | 1 | SC | RL | - | - | SC=1:display, 0:cursor<br>RL=1:right, 0: left shift |
| **Function set** | 0 | 0 | 1 | DL | N | F | - | - | DL=1:8-bit, 0:4-bites mode<br>N=1:2 sor, 0:1 sor<br>F=1:5x10, 0:5x8 pixels |
| **Set CGRAM address** | 0 | 1 | | | | | | | Address of writing to the character generator RAM, defining characters |
| **Set DDRAM address** | 1 | 0 | | | | | | | Address of writing to a specific location of the display, cursor positioning |

When writing to or reading from the display, certain timing conditions must be met (see Figure 4.12). The display is a rather slow external peripheral, and the microcontroller code must be written with this taken into account.

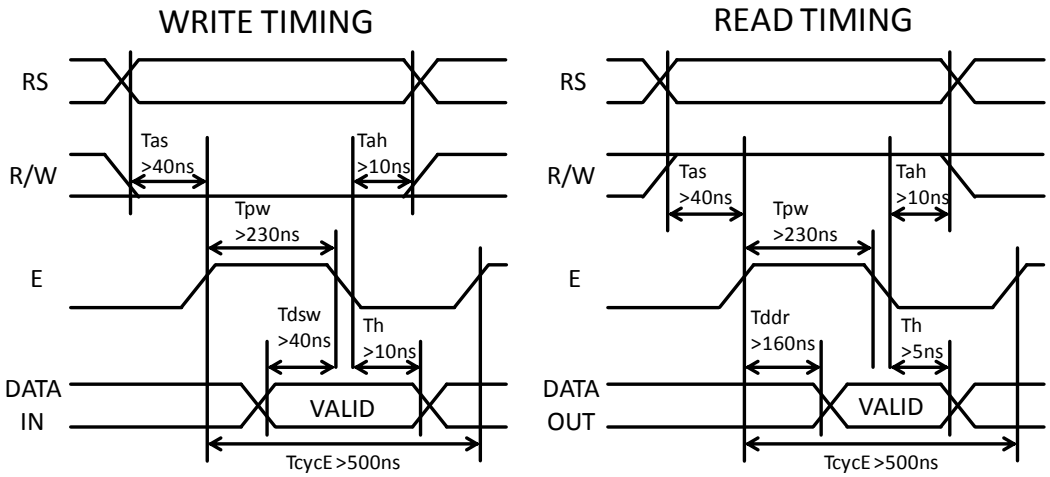


*Figure 4.12. Time diagram of write and read operations (for details see the HD44780 datasheet [16]).*

The following example code introduces a few functions to initialise the display and to write to the display.

```
#define LCD_RS   P0_5  // RS is the register select input
#define LCD_RW   P0_6  // specifies read or write
#define LCD_E    P0_7  // enable line serves as write or read pulse
#define LCD_PORT P1    // the data bits are connected to port P1

unsigned char line_address[4];  // this array holds the address of the
                                // first character in a row

/*********************************************************************
LCD initialisation function
Input parameters are the number of rows and columns
*********************************************************************/
void LCD_Init(unsigned char rows, unsigned char columns)
{
    unsigned char i;
```

```
    line_address[0]=0;     // initial address of the first row
    line_address[1]=0x40;  // initial address of the second row
    line_address[2]= columns;      // initial address of the third row
    line_address[3]=0x40+ columns; // initial address of the fourth row
    LCD_RW=0;         // assume write operations as default
    LCD_E=0;          // the E line should be inactive
    LCD_RS=0;         // register select must be 0 to send commands
    Delay_ms(50);     // special initialisation sequence after 50 ms of delay
    LCD_DATA=0x30;    // 8-bit mode is selected
    LCD_PulseE();     // generate pulse on the E line
    Delay_ms(5);      // wait for approximately 5 ms
    LCD_PulseE();     // generate pulse on the E line
    Delay_ms(1);      // wait for approximately 1 ms
    LCD_PulseE();     // generate pulse on the E line
    LCD_Write(0x38); // set 8-bit mode, 2 lines
    LCD_Write(0x08); // set display off
    LCD_Write(0x01); // execute display clear function
    LCD_Write(0x06); // set entry mode: increment cursor
    LCD_Write(0x0C); // switch display on; no cursor; no blinking selected
}

/************************************************************************
Pulses the E line to initiate a write to or read from the LCD
************************************************************************/
void LCD_PulseE(void)
{
    __asm             // wait for about 1 µs
        mov R7,#7     // system clock frequency in MHz divided by 4
    L1: djnz R7,L1    // loop to wait for the specified time
    __endasm;
    LCD_E=1;          // set E
    __asm             // wait for about 1 µs
        mov R7,#7     // system clock frequency in MHz divided by 4
    L2: djnz R7,L2    // loop to wait for the specified time
    __endasm;
    LCD_E=0;          // clear E
}

/************************************************************************
Writes a byte to the LCD
************************************************************************/
void LCD_Write(unsigned char a)
{
    LCD_DATA=a;       // set the data bus according to the value of a
    LCD_PulseE();     // generate pulse on the E line
    Delay_ms(2);      // wait 2 ms here or, alternatively, check busy flag
}

/************************************************************************
Clears the entire LCD display
************************************************************************/
void LCD_Clear(void)
{
    LCD_RS=0;         // register select must be 0 to send commands
    LCD_Write(1);     // write command to LCD
    LCD_RS=1;         // register select default value is 1 (display RAM)
}
```

```
/*************************************************************************
Moves the cursor to the specified location
*************************************************************************/
void LCD_MoveTo(unsigned char line, unsigned char pos)
{
    LCD_RS=0;        // register select must be 0 to send commands
    LCD_Write(0x80 | (line_address[line]+pos));  // select position of char
    LCD_RS=1;        // register select default value is 1 (display RAM)
}

/*************************************************************************
Redirects the standard C output to the LCD
*************************************************************************/
void putchar(char c) // redefined standard output function
{
    LCD_Write(c);    // send the character to the LCD
}

LCD_MoveTo(0,10);    // first line, 10th position
printf("Hello");     // write to the display
```

### *4.3.6 Driving relays and motors*

Microcontrollers must sometimes control higher power devices such as motors, stepper motors, valves or high power LEDs. Since the output can source and sink only a few milliamperes, external drivers are required for heavy loads. A simple solution is to use bipolar or MOS transistors connected to port pins configured as push-pull, as shown in Figure 4.13. Inductive loads – coils or motors – can cause very high voltage spikes during turn-off; therefore, a protection diode is used across the two terminals of such a load.

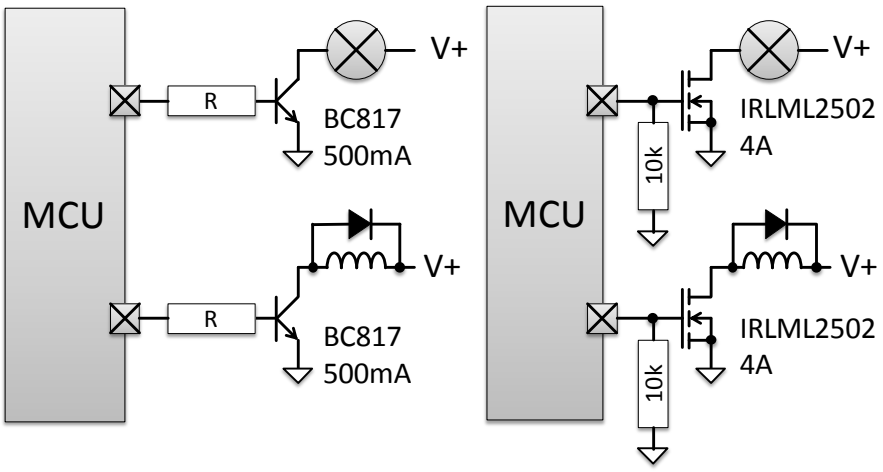


*Figure 4.13. Higher power loads can be driven by external transistors.*

## *4.4 Application guidelines*

- The crossbar must be enabled to connect the port bits to the pins.
- Pins used as digital inputs should be configured as open-drain and logic high must be written to the corresponding port bits.
- After reset, the port latches are set to 1.
- Pins used as digital outputs should be configured as push-pull.

- The sink current (open-drain or push-pull mode) or source current (push-pull mode only) must not exceed the datasheet specifications (about 3 mA-4 mA). The total current of all output pins should be limited to meet the specifications.
- An LED can be connected to an output via a series current limiting resistor (330 Ω–1 kΩ). If using open drain output, the cathode must be connected to the supply voltage.
- Buttons can be connected between the input and GND. An optional pull-up resistor of about 10 kΩ can be used. Parallel capacitors may help to reduce switching noise.

## *4.5 Troubleshooting*

**Problem:**

- Cannot read from or write to the port pin; unexpected output or input values experienced.

**Possible reasons:**

- The crossbar is disabled. Enable the crossbar.
- Writing 1 to a port bit does not mean that reading it returns 1, since the voltage can be pulled low by an external circuit.
- Push-pull is not set for an output. Open-drain output cannot source current.
- Be sure that for input the port pin is configured as open-drain and 1 is written to the port bit.
- External short-circuit on the pin can be present. Check the voltage with a voltmeter.
- Improper logic voltage level may be present on the pin.
- The output drive current limit may be violated; too high a load may be present on the pin.
- A peripheral is associated with the port pin by the crossbar. In this case the pin state can only be driven by the peripheral.
- Short glitches (temporary pulses) on a signal can corrupt reading.

## *4.6 Exercises*

- *Write a program that reads the state of a button and toggles the state of an LED on each pressing. Successive button pressings within 200 ms should not be detected. Continuously pressing a button should be detected as a single pressing.*
- *Connect four LEDs and a button to the microcontroller and write code that illuminates only one LED and switches to the next LED if the button is pressed. The first LED should follow the last one.*
- *Write code that displays incremented numbers from 0 to 9 in a cyclic manner upon each pressing of a button.*
- *Write a program that detects the first button pressing for a longer time and halves the detection time for additional detections if the button is continuously pressed.*
- *Change the code to reduce detection period if the button is continuously pressed.*
- *Write code to read which key is pressed on a 3 × 4 keyboard matrix. Flash an LED as many times as the number corresponding to the key pressed.*
- *Write code to display the number of button pressings on an LCD display.*

- *Connect a unipolar stepper motor to four pins of the microcontroller. Write a program that energises only one coil at a time and switches to the next coil ten times per second.*
- *Modify the program to switch between rotating clockwise and anticlockwise directions when another button is pressed.*

# 5 Timers and counters

The basis of the timers integrated into microcontrollers is a binary counter. It realises a timer function if it is driven by an oscillator. It can be used to count events corresponding to falling or rising edges of an external logic signal.

## 5.1 Timer 0 and Timer 1

C8051Fxxx processors contain enhanced versions of the standard 16-bit 8051 timers Timer 0 and Timer 1 [6]. Their clock input can be configured as shown in Figure 5.1. SYSCLK is the system clock; EXT OSC represents an external oscillator. In timer mode, the TCLK signal driving the timer clock input is derived from these sources, while in counter mode the T0 input is used. The **TR0** SFR bit enables the timer and the /INT0 input can be used to gate the timer depending on the state of the **IN0PL** and **GATE0** SFR bits. The timer/counter values can be accessed via the SFRs **TH0** and **TL0** as well as **TH1** and **TL1,** representing the higher and lower order bytes of Timer 0 and Timer 1, respectively. If the timer overflows, a flag is set (**TF0** for Timer 0 and **TF1** for Timer 1) and an interrupt can be generated if enabled.

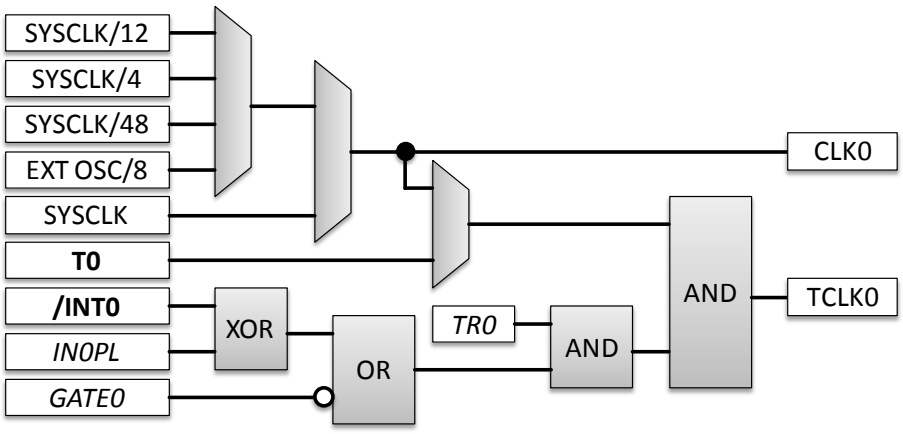


*Figure 5.1. Timer input clock configuring circuit. Labels in bold indicate external signals and configuration bits are in italic.*

Timer 0 and 1 have four modes of operation. In mode 0, they are operated as 13-bit timers; only the lower 5 bits of the lower order byte of the timers are used (see Figure 5.2).


*Figure 5.2. Timer 0 in mode 0 (13-bit timer) and in mode 1 (16-bit timer). Each falling edge on **TCLK0** increments the timer.*

In mode 2, only the lower 8 bits are used for counting and the higher order byte is used as a starting value upon overflow. This is called auto reload mode and is useful for generating programmable periodic events. The block diagram of this mode is illustrated in Figure 5.3.

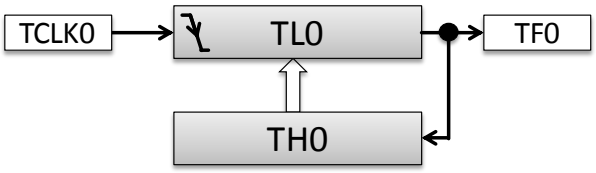


*Figure 5.3. Timer 0 in auto reload mode. Upon overflow, the initial value is loaded to **TL0** from **TH0**.*

The time that passes from the initial state (when the counter value **TL0** is equal to **TH0**) to the overflow is

$$T = (256 - TH0) \cdot \Delta t_{\mathrm{TCLK0}}, \tag{5.1}$$

where $\Delta t_{\mathrm{TCLK0}}$ is the period of the input clock of the timer.

The frequency of the periodic overflows can be given by

$$f = f_{\mathrm{TCLK0}} \frac{1}{256 - \mathrm{TH0}}, \tag{5.2}$$

where $f_{\mathrm{TCLK0}}$ is the frequency of the input clock of the timer

If the initial value of the counter (**TL0**) is less than the reload value, an overflow must occur before generating the overflows with the desired rate. The mechanism of the auto reload mode is illustrated in Figure 5.4.

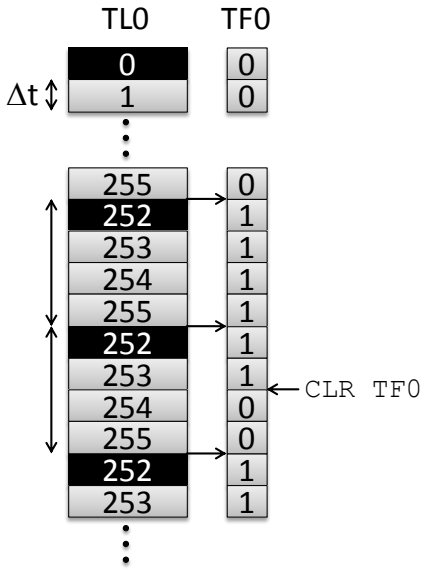


*Figure 5.4. Timer 0 operation in auto reload mode.*

Mode 3 is rarely used; here, the two 8-bit parts of Timer 0 are used as two 8-bit timers as shown in Figure 5.5. Timer 1 is inactive in this mode.

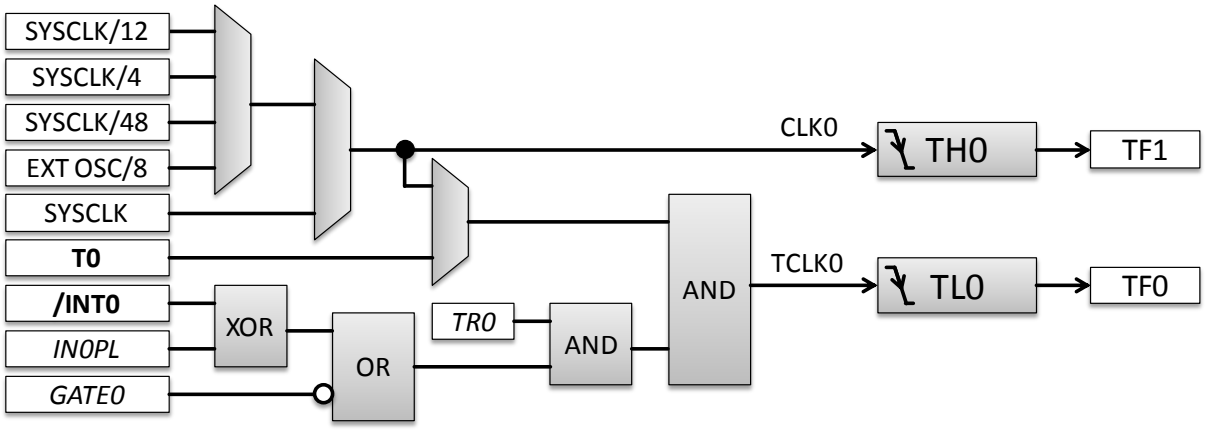


*Figure 5.5. In mode 3, two 8-bit timers can be used. Labels in bold indicate external signals and configuration bits are in italic.*

## 5.2 Timer 2, Timer 3 and Timer 4

C8051Fxxx processors have 2, 3 or 4 additional 16-bit timers with various features [6].

The timers of the C8051F410 can be operated as 16-bit auto reload timers or as dual 8-bit auto reload timers.

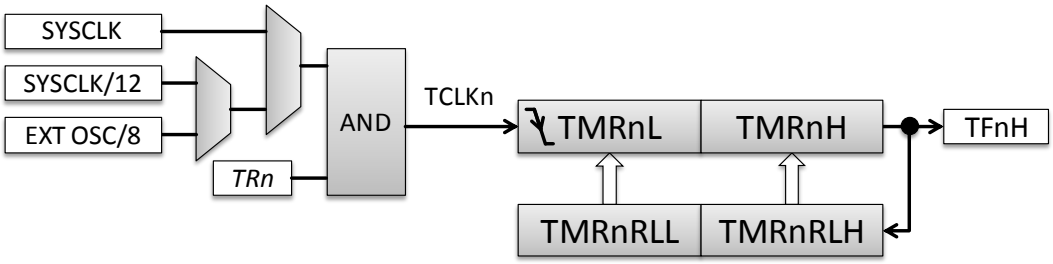


*Figure 5.6. Timers 2, 3 and 4 in 16-bit auto reload mode. The label in italic is used for the timer enable bit.*

Figure 5.6. shows the block diagram of the auto reload operation, while the operation itself is illustrated in Figure 5.7.

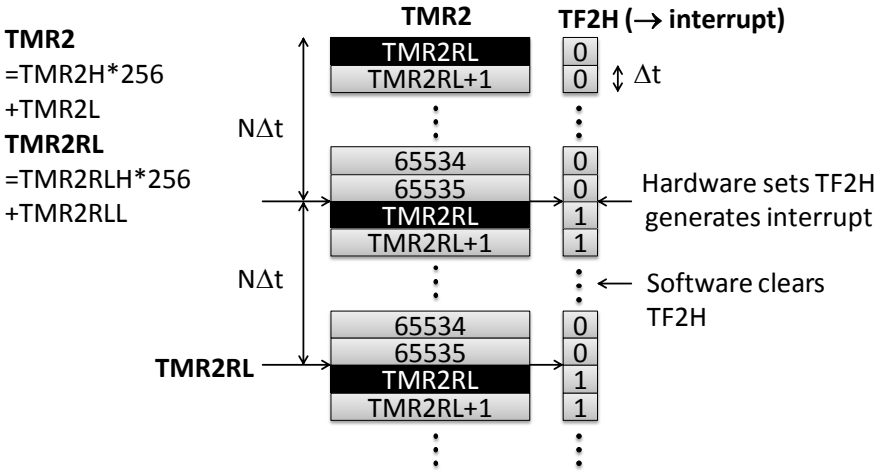


*Figure 5.7. Timer operation in 16-bit auto reload mode.*

Using Timer 2 the frequency of the periodic overflows is given by

$$f = f_{\mathrm{TCLK2}} \frac{1}{65536 - \mathrm{TMR2RL}} = f_{\mathrm{TCLK2}} \frac{1}{65536 - 256 \cdot \mathrm{TMR2RLH} + \mathrm{TMR2RLL}}; \qquad (5.3)$$

therefore, if the desired frequency is known, the value of the reload registers can be calculated:

$$\begin{aligned} \mathrm{TMR2RL} &= 65536 - \frac{f_{\mathrm{TCLK2}}}{f}, \\ \mathrm{TMR2RLH} &= \left\lfloor \frac{\mathrm{TMR2RL}}{256} \right\rfloor, \\ \mathrm{TMR2RLL} &= \mathrm{TMR2RL} \bmod 256 \end{aligned} \qquad (5.4)$$

The following example code illustrates the calculation of the reload value. The desired period is given in μs units; the timer input clock must be entered in Hz units.

```
/**************************************************************************
steps = period/dt
dt=1/timer clock
steps = timer clock*period
reload value = 65536-steps
**************************************************************************/

unsigned long period;  // in µs
unsigned long tmrclk;  // timer clock in Hz
unsigned short tmrrl;  // reload value

tmrclk = 24500000;              // timer clock frequency is 24.5 MHz
period = 100;                   // 100 µs
tmrrl = -period*tmrclk/1000000L; // means 65536-period*tmrclk/1000000L
```

Note that it is also possible to configure the timer as two 8-bit auto reload timers; see Figure 5.8.

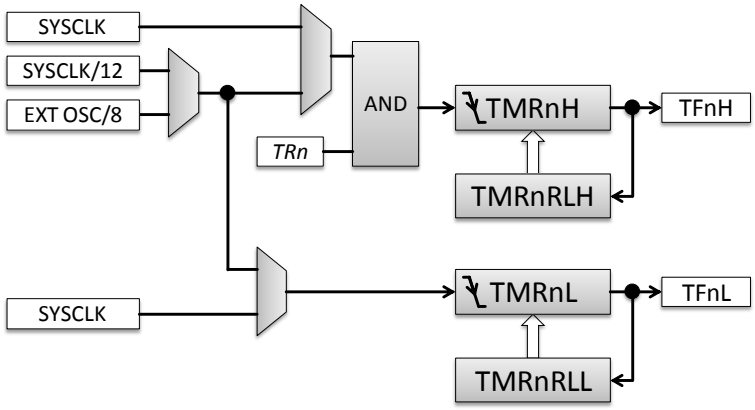


*Figure 5.8. Timers 2, 3 and 4 in C8051F410 can be configured as two 8-bit auto reload timers. The label in italic is used for the timer enable bit.*

An enhanced timer is available in some C8051Fxxx processors, including, for example, the C8051F120 100-MHz microcontrollers. As illustrated in Figure 5.9, this timer can count up or down, and can be clocked by an external signal Tn. The external signal *TnEX* can be used to

latch the counter value into the reload registers, which allows the accurate detection of the time instant of an event or a series of events.

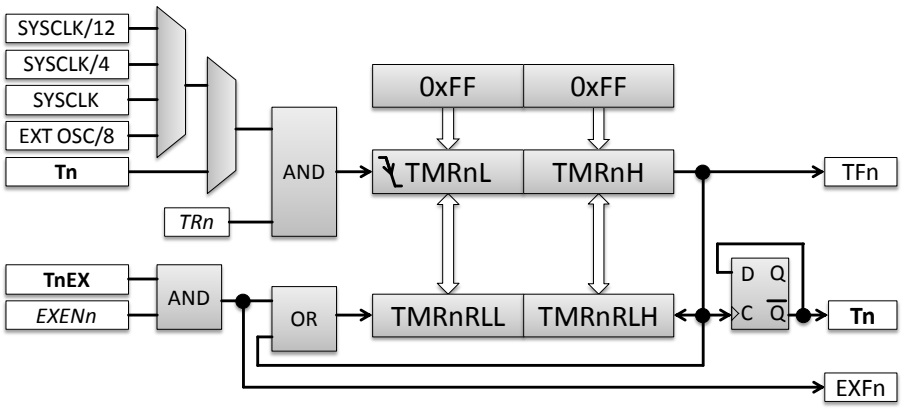


*Figure 5.9. Timers 2, 3 and 4 in C8051F120 can count up or down, toggle an output signal and capture the timer value. Labels in bold indicate external signals and italic is used for configuration bits.*

## 5.3 Timer applications

Timers can be used in many applications. They are employed to generate periodic interrupts, to provide programmable clock frequency for serial communication ports and also to generate events at certain time instants and to measure the time elapsed between events.

Some examples are given below.

### 5.3.1 Delay generation

Timers can be used to generate a desired amount of time delay. The time required to reach the overflow state from the initial value of Timer 0 (**TH0**, **TL0**) can be given by the following formula:

$$T = \left(65536 - \left(\mathrm{TH0} \cdot 256 + \mathrm{TL0}\right)\right) \cdot \Delta t_{\mathrm{TCLK0}}, \tag{5.5}$$

where $\Delta t_{\mathrm{TCLK0}}$ is the period of the input clock of the timer. The code example below implements a function that waits for a period equal to *steps*·$\Delta t_{\mathrm{SYSCLK}}$.

```
/*************************************************************************
Waits for a specified number of Timer 0 steps
*************************************************************************/
void Delay(unsigned short steps)
{
    TMOD=(TMOD & 0xF0) | 0x01;  // 16-bit timer mode
    CKCON=CKCON | 0x04;         // Timer 0 clock is SYSCLK
    TH0=-steps >> 8;            // 65536-steps, higher-order byte
    TL0=-steps;                 // 65536-steps, lower-order byte
    TF0=0;                      // clear timer overflow flag
    TR0=1;                      // run Timer 0
    while (!TF0);               // wait for Timer 0 to overflow
    TR0=0;                      // stop Timer 0
}
```

### 5.3.2 Generating periodic interrupts

Periodic interrupt generation is a common application. Timer auto reload mode is one option. The following C8051F410 code is the C version of the code given in Chapter 3.2:

```
/************************************************************************
Timer 2 interrupt handler routine
************************************************************************/
void IntHandler(void) __interrupt 5  // define Timer 2 interrupt handler
{
    TF2H=0;     // clear interrupt pending flag
    LED = !LED; // complement LED; flashing rate is half
                // of the Timer 2 overlow rate
}

/************************************************************************
The main function
************************************************************************/
void main(void)
{
    PCA0MD &= 0x40h; // switch watchdog off
    PCA0MD  = 0x00h; // switch watchdog off
    XBR1    = 0x40h; // enable the crossbar to allow input and output
    TMR2RLL = 0xB2h; // set the Timer 2 reload register (low and high bytes)
    TMR2RLH = 0xC1h; // to provide 1-Hz interrupt rate
    TMR2L   = 0xB2h; // Timer 2 counter initial value
    TMR2H   = 0xC1h; // is the same as the reload value
    TMR2CN  = 0x04;  // start Timer 2 now
    IE      = 0xA0;  // enable global interrupts and Timer 2 interrupt
    while (1);       // infinite loop
}
```

Timer 0 and Timer 1 provide 8-bit auto reload mode; therefore, only higher frequencies can be generated. In the example below, the program sets the initial value upon overflow. Note that due to the latency time it is not as accurate as the hardware auto reload mode.

```
TMOD=(TMOD & 0xF0) | 0x01;  // 16-bit timer
TR0=1;     // run timer
IE=0x82;   // enable global & timer0 interrupts

/************************************************************************
Timer 0 interrupt handler routine
************************************************************************/
void Timer0Handler(void) __interrupt 1  // define Timer 0 interrupt handler
{
    TR0=0;            // stop timer
    TH0=-steps >> 8;  // initial value is 65536-steps (higher-order byte)
    TL0=-steps;       // 65536-steps (lower-order byte)
    TR0=1;            // restart timer
    …                 // perform the required operation
}
```

### 5.3.3 Software extended counter

The 16-bit counter can easily be extended by software. For example, if a 24-bit counter is needed, an 8-bit variable can be added to represent the most significant 8-bits while the hardware timer provides the least significant 16-bits. Upon overflow of the timer the variable is incremented as shown in Figure 5.10.

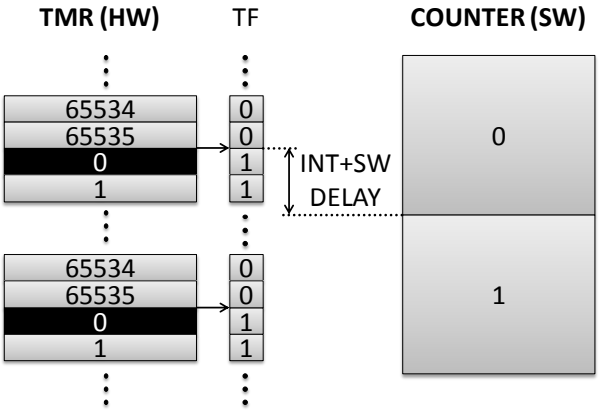


*Figure 5.10. 24-bit timer operation emulated by software.*

```
/************************************************************************
Timer interrupt handler routine
************************************************************************/
void TimerIRQ(void) __interrupt TIMER_VECTOR // define interrupt handler
{
    static unsigned char counter=0;  // static variable retains value

    counter = (counter+1) % countermax; // increment and implement overflow
    if (!counter) Process();            // if counter returns to zero
                                        // (overflows)
}
```

### *5.3.4 Pulse width measurement*

Timers can be used to measure event timing. Figure 5.11 shows the time diagram of a possible pulse width measurement. The /INT0 external signal is used to gate the timer: the timer counts while this signal is high. In order to set up properly, the code waits first for /INT0 to go low, then enables the timer. After this, the code should wait for the next falling edge of /INT0, which identifies the end of the pulse. Note that the /INT0 state or a falling edge can set the **IE0** flag, which can be polled or used to generate an interrupt. /INT0 must be enabled using the crossbar.

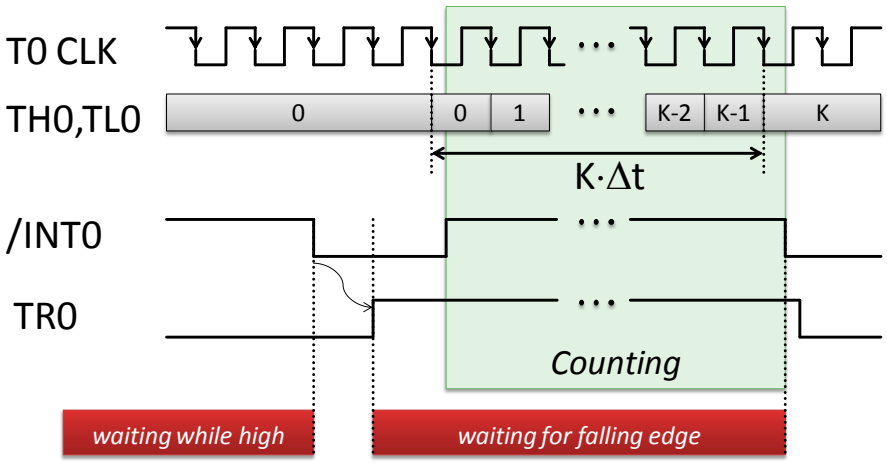


*Figure 5.11. Time diagram of the pulse width measurement of the /INT0 signal.*

```
TR0=0;         // stop timer
TH0=TL0=0      // clear timer
TMOD=0x09;     // T0: 16-bit gated timer mode
IT0=0;         // set level-triggered /INT0 mode (IE0 is high if /INT0 is low)
IE0=0;         // clear INT0 flag
while (!IE0); // wait for input to go down
IT0=1;         // set edge-triggered /INT0 mode to detect falling edge
IE0=0;         // clear INT0 flag
TR0=1;         // enable timer
while (!IE0); // wait for end of pulse
TR0=0;         // stop timer
```

The result of the measurement is in the registers **TH0** and **TL0**.

Note that since part of the detection is done by software, the accuracy can be affected by accidental interrupts, which halt the main code for a while.

### *5.3.5 Frequency measurement*

The time diagram of a possible frequency measurement algorithm can be seen in Figure 5.12. The timer is enabled for a given amount of time (for example 1 s) and the counter counts the external signal falling edges. Therefore, the frequency is the counter value divided by the running time.

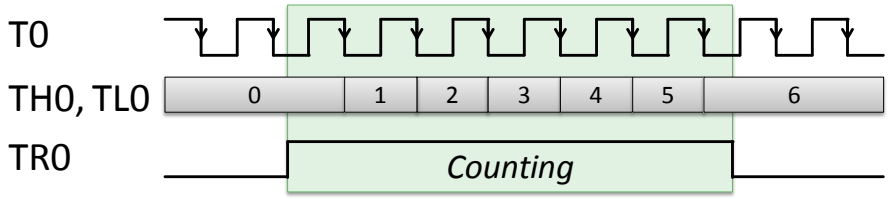


*Figure 5.12. Time diagram of the frequency measurement.*

In the following example, Timer 1 is used to set **TR0** high for a given amount of time, while Timer 0 counts the pulses.

```
TCON=0;          // stop Timer 0 and Timer 1
TMOD=0x15;       // Timer 0: 16-bit counter; Timer 1: timer
TH0=0;           // initialise counter of Timer 0 (high byte)
TL0=0;           // initialise counter of Timer 0 (low byte)
TH1=-steps >> 8; // 65536-steps (high byte)
TL1=-steps;      // 65536-steps (low byte)
TF1=0;           // clear Timer 1 overflow flag
TCON=0x50;       // run both timers (both TR0 and TR1 are set)
while (!TF1);      // wait for Timer 1 overflow (Timer 0 counts during this
time)
TCON=0;          // stop both timers
```

The result of the measurement is in the registers **TH0** and **TL0**.

Note that since part of the detection is done by software, the accuracy can be affected by accidental interrupts, which halt the main code for a while.

### 5.3.6 Period measurement

Period measurement means that the time of one or more periods is measured. If the period is short, it is better to measure the time of multiple periods. One timer can be used to count the periods; its initial value must be set to 65536 minus the number of periods to be counted. The other timer is driven by a clock source and runs while the first counter is counting. Therefore, the time of one period is the clock period of the second timer multiplied by the clock period and divided by the number of pulses counted.

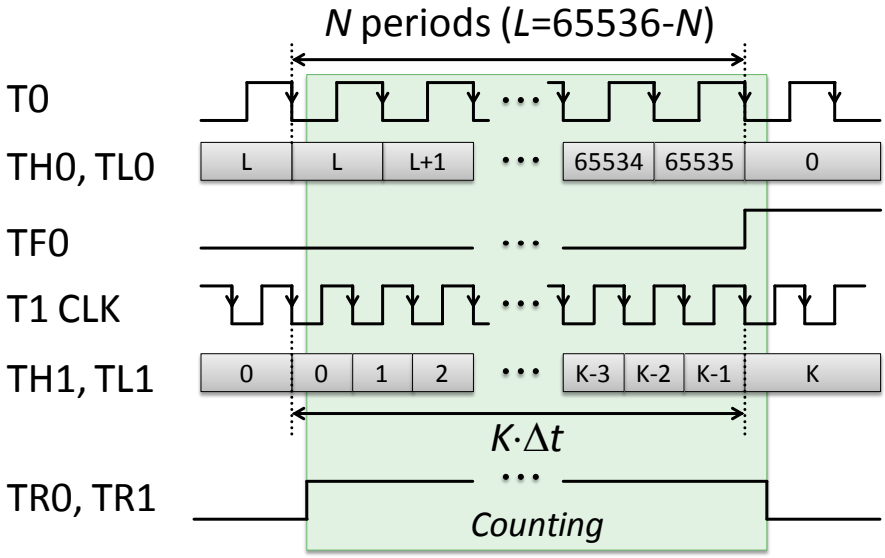


*Figure 5.13. Time diagram of the period measurement.*

```
TCON=0;       // stop timers
TMOD=0x15;    // T0: 16-bit counter; T1: timer dTCLK1 period
TH1=0;        // timer1 initial value (high byte)
TL1=0;        // timer1 initial value (low byte)
TH0=-N>> 8;   // 65536-N
TL0=-N;       // N events to TF0=1
TF0=0;        // clear timer 0 flag
TCON=0x50;    // run both timers
while (!TF0); // wait for N events
TCON=0;       // stop both timers
```

The result of the measurement is in the registers **TH1** and **TL1**. The period is:

$$T = \frac{(\mathrm{TH1} \cdot 256 + \mathrm{TL1}) \cdot \Delta t_{\mathrm{TCLK1}}}{N}, \tag{5.6}$$

Note that since part of the detection is done by software, the accuracy can be affected by accidental interrupts, which halt the main code for a while.

## 5.4 Application guidelines

- The timer input clock must be configured first. Choose a frequency value that allows the desired rate to be accurately set. If the clock frequency is too high, the overflow rate cannot be set low enough or longer time intervals cannot be measured. If it is too low, the accuracy of the timing can be low.

- Verify the settings by calculating the timing using the timer SFR values.
- Set the desired operating mode.
- After proper setup, enable the timer.
- Enable the timer interrupt if needed. Do not forget to clear the interrupt pending flag in the service routine.
- The timer interrupt must not be enabled if no service routine is defined. This is the case when the timer is used to generate a periodic signal for other communication or other peripherals (UART, PCA, etc.)
- Keep in mind that using the timer for multiple purposes simultaneously needs special attention and is a potential source of problems.
- Do not read 16-bit timer values during operation, since the high and low bytes cannot be read simultaneously and thus they may not correspond to the same timer value.

## *5.5 Troubleshooting*

**Problem:**

- The timer is not running or unexpected timing occurs.

**Possible reasons:**

- The timer is not enabled.
- The timer is not configured in the proper mode.
- The input clock is not configured properly.
- Timer 0 and Timer 1 may be in gated mode and the gate signal may be inactive.
- The SFR values are miscalculated or not properly written.

**Problem:**

- No timer interrupt occurs or the interrupt rate is not as expected.

**Possible reasons:**

- The timer is not enabled.
- The associated interrupt is not enabled.
- The interrupt flag is not cleared, so the interrupt is generated continuously. Most of the processor power is taken in this case.
- Execution of other interrupt service routines can cause a delay of the timer interrupt.
- The service routine may take longer than the time between two overflows; the overflow rate is too high.
- The timer is used for multiple purposes simultaneously and the settings are different.

**Problem:**

- Unexpected frequency, period or pulse width of an external signal is experienced.

**Possible reasons:**

- The crossbar is not configured properly to connect the external signal to the timer.
- The timer settings (like input clock or mode) are improperly set or miscalculated.
- The resolution of the time measurement is too low; for example, short periods are measured by only a few timer increments.
- The external signal is noisy; oscillations occur at transitions.

- The software-dependent part of the measurement code is delayed by an unexpected interrupt.

## *5.6 Exercises*

- *Write a function that waits for a specified number of milliseconds given by the argument of the function. Use Timer 0 to implement the function.*
- *Write a program that flashes an LED at 1 Hz. Use a button to set the flashing rate: on each pressing of the button, the rate should be doubled, but at 16 Hz the pressing of the button should reset the rate to 1 Hz.*
- *Write a program that emulates a pulse width modulated signal. An LED should be flashed at a rate of 100 Hz and the on time should be set by button pressings from 1 ms to 9 ms in a cyclic manner.*
- *Write a program that generates an output signal of 100 Hz using Timer 2. Measure the period of this signal using the other timers.*
- *Connect a 555 timer circuit based 100 Hz oscillator output to the microcontroller. Measure the frequency of this signal using timers.*
- *Measure the pulse width of button pressings using timers.*

# 6 Programmable counter array

The programmable counter array (PCA) contains a simple 16-bit free-running counter, which is driven by a periodic clock signal [6, 17]. There are several (from 3 to 6) independent compare/capture registers, which can be used to latch the counter value upon an event (change in a digital input signal). These registers can also hold data to be compared with the counter value and to generate an event when a match occurs. The corresponding flag (**CCFn**) is set upon these events, while the **CF** flag is set when the main counter overflows. All of these events can generate interrupts; however, the same interrupt routine is called, so the flags must be checked to identify the source of the interrupt. The structure of the PCA is shown in Figure 6.1.

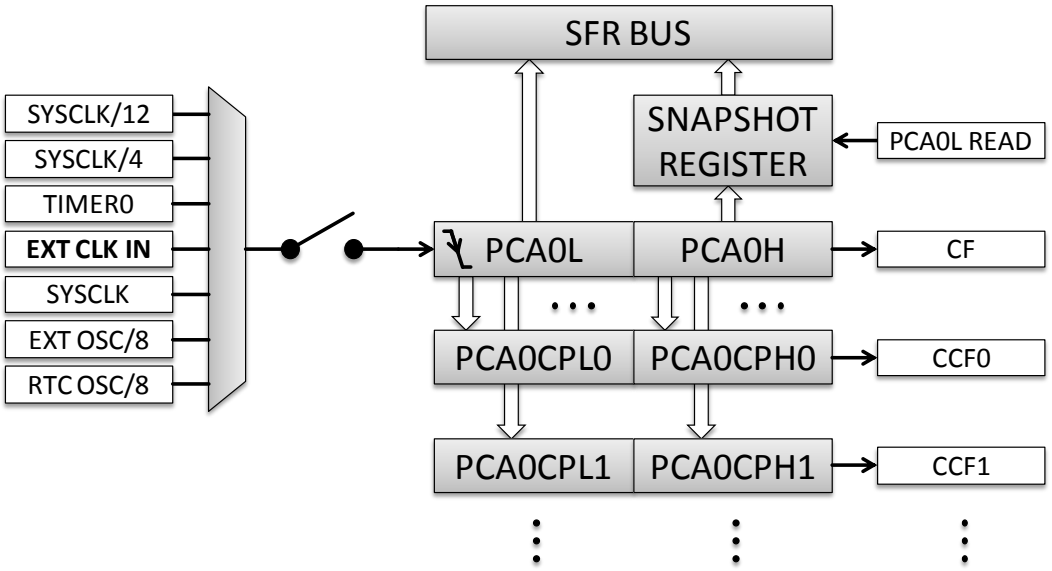


*Figure 6.1. The main counter (**PCA0L** and **PCA0H**) can be driven from different clock sources. There are up to six compare/capture registers (**PCA0CPLn** and **PCA0CPHn**). The label in bold indicates an external signal.*

Depending on the operating mode, several useful functions can be implemented.

In the following, only one of the six compare/capture registers is shown, and the names of the corresponding SFRs are appended with an **n** that identifies one of the six possible registers.

Note that all compare/capture registers can be associated with an external input/output signal named **CEXn** (**n** = 0, 1, ...) via the crossbar. The function of this signal is determined by the operation mode.

## 6.1 Edge-triggered capture mode

The edge-triggered capture mode uses an external signal **CEXn** to latch the value of the counter into one of the capture registers (**PCA0CPLn** or **PCA0CPHn**). This can happen on rising or falling transitions, or on both. The **CCFn** flag is set and an interrupt can be generated, if enabled. Figure 6.2 shows the block diagram of the edge-triggered capture mode.

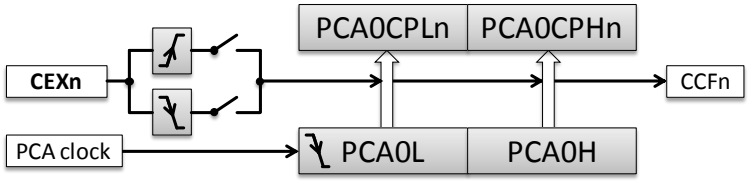


*Figure 6.2. Edge-triggered capture mode. The label in bold indicates an external signal.*

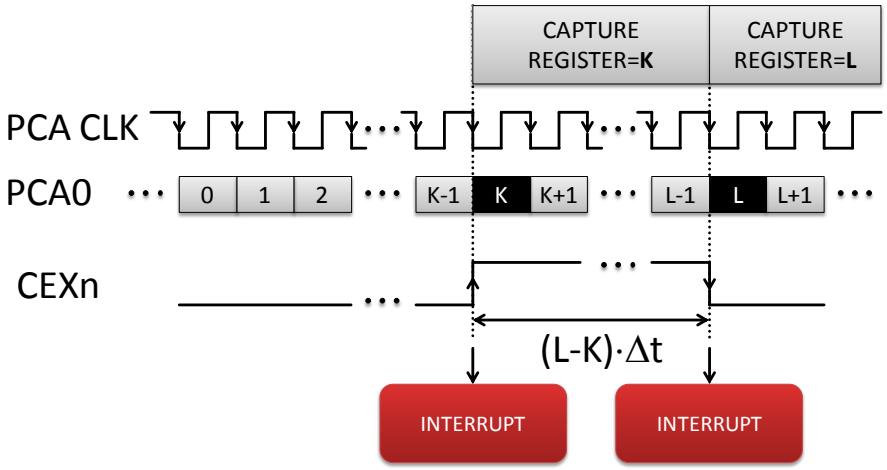


*Figure 6.3. Time diagram of the edge-triggered capture mode.*

## 6.2 Software timer and high-speed output mode

The **PCA0CPLn** and **PCA0CPLh** registers can be compared to the actual value of the main counter, setting the **CCFn** flag and generating an interrupt if a match occurs. The software timer and the high-speed output mode are practically the same, except that in the output mode the **CEXn** output is toggled upon each match event. Figure 6.4 shows the block diagram of these modes. Note that a write to **PCA0CPLn** disables the comparator, while writing **PCA0CPHn** enables it. This ensures that both the low and the high byte of the capture/compare register are valid when the comparator is enabled. The programmer must take it into account, so **PCA0CPLn** *must be written first and then should the value of* **PCA0CPHn** *be set*. Changing only **PCA0CPLn** stops the operation.

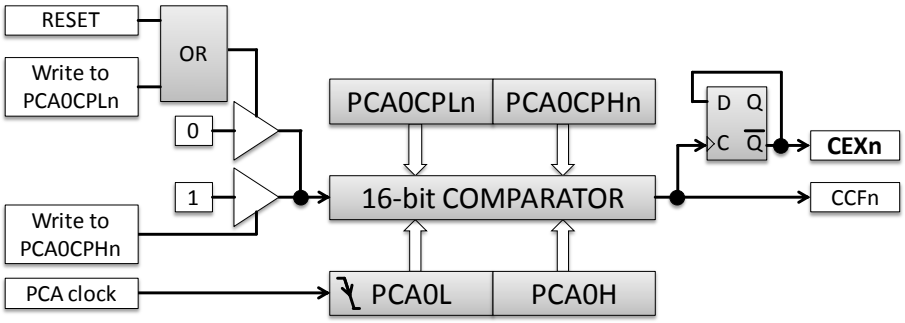


*Figure 6.4. Software timer and high-speed output mode. The label in bold indicates an external signal.*

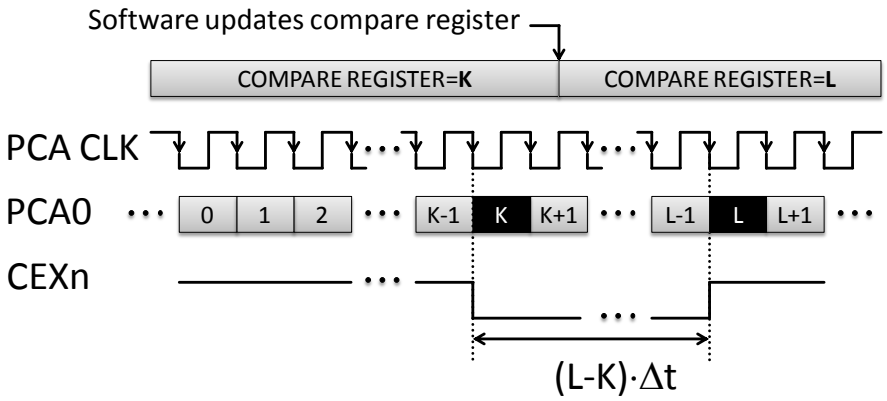


*Figure 6.5. Time diagram of the software timer and of the high-speed output mode.*

## 6.3 Frequency output mode

Frequency output mode (Figure 6.6) can be used to output a periodic square wave. Only the 8 least significant bits of the counter are compared to **PCA0CPLn** and upon a match the output is toggled and the **PCA0CPLn** is incremented by the value stored in **PCA0CPHn**. Of course, **PCA0CPLn** will overflow at a certain time but it does not affect the operation.

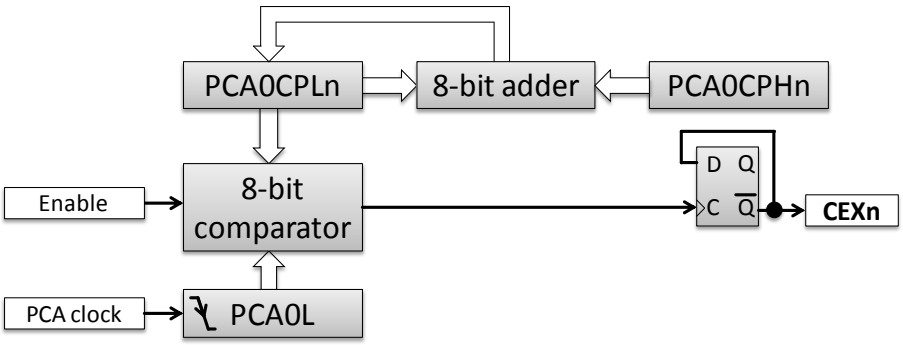


*Figure 6.6. Frequency output mode.*

$$f = \frac{f_{PCA}}{2 \cdot PCA0CPHn} \tag{6.1}$$

Figure 6.7 shows a sample time diagram when the output frequency is $f_{PCA}/6$.

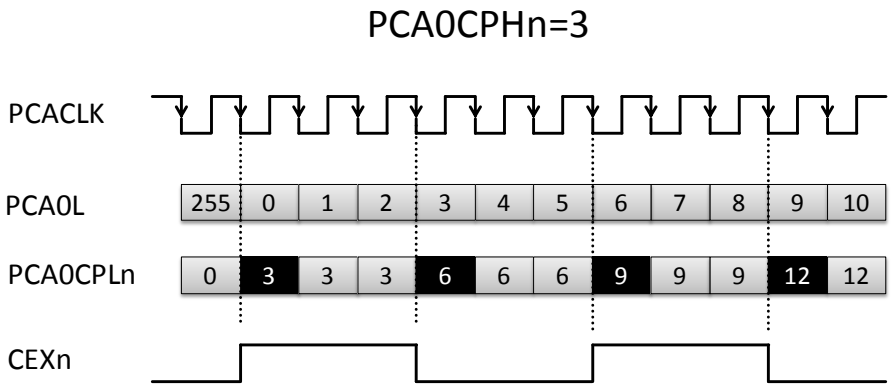


*Figure 6.7. Time diagram of the frequency output mode when the frequency is 1/6 of the PCA input frequency.*

## 6.4 8-bit and 16-bit PWM modes

One of the most useful modes is the generation of pulse width modulated (PWM) signals. Since a single digital signal can only have two different values, its applications in control are strictly limited. Using PWM signals, this limitation can be significantly reduced.

A PWM signal is a periodic pulse train whose pulse width can be varied. If the frequency of this signal is high enough, it can be used as a fine control of slow systems. Typical applications include motor control, temperature control and light control, where the driven system cannot follow fast changes and thus only the average of the signal will be effective. This average is proportional to the duty cycle of the PWM signal.

The PCA module supports 8- and 16-bit PWM modes. In the 8-bit mode, only the 8 least significant bits of the counter are used. When the value is equal to **PCA0CPLn**, the output signal is set, and at the overflow of **PCA0L,** it will be reset; see Figure 6.8. This way, the signal is low for **PCA0CPLn** steps and high for 256-**PCA0CPLn** steps. This can be changed by writing a new value to **PCA0CPHn**, which will take effect only upon the overflow of **PCA0L**, ensuring reliable changes. The frequency of the PWM signal is $f_{PCA}/256$.

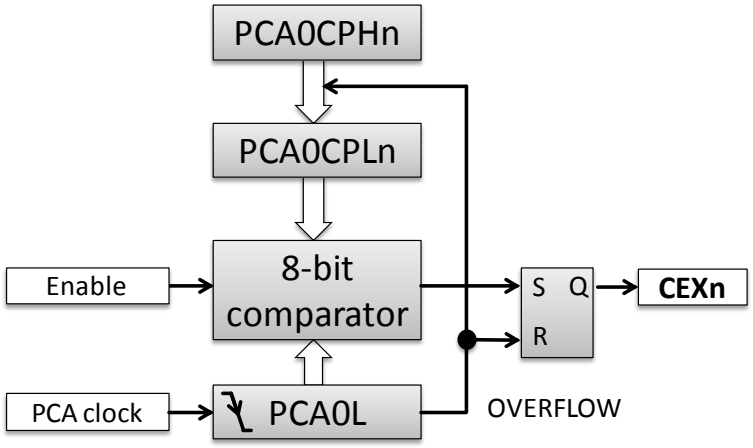


*Figure 6.8. 8-bit PWM mode. The label in bold indicates an external signal.*

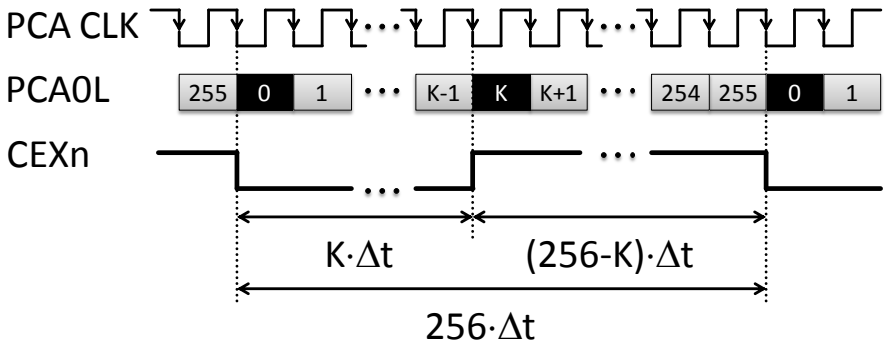


*Figure 6.9. Time diagram of the 8-bit PWM mode.*

The 16-bit PWM operation is similar, but here all 16 bits of the counter as well as compare registers are used; see Figure 6.10 and Figure 6.11.

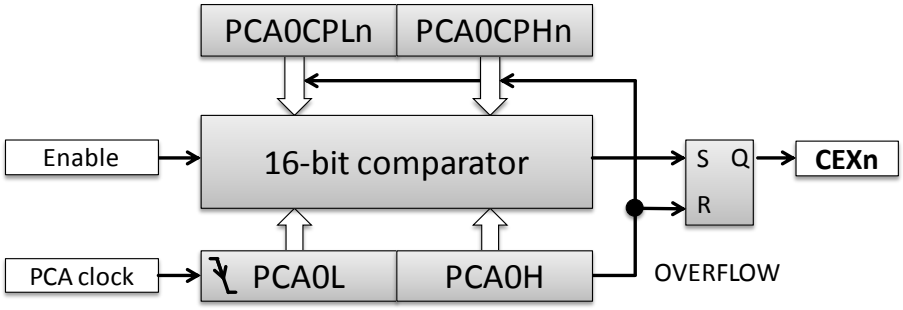


*Figure 6.10. 16-bit PWM mode. The label in bold indicates an external signal.*

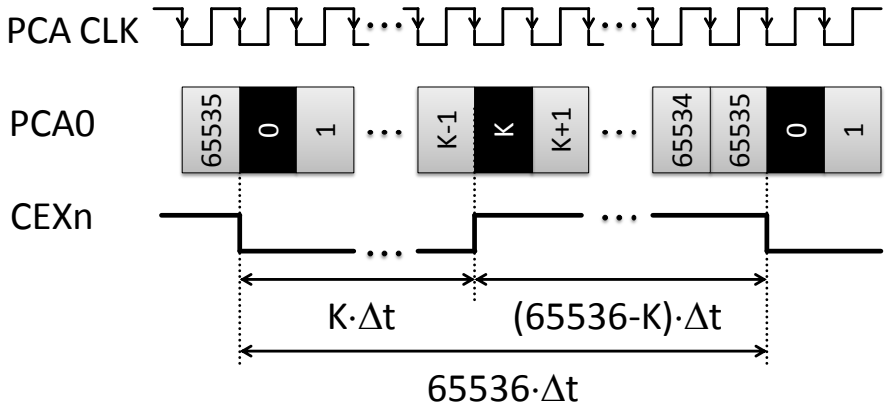


*Figure 6.11. Time diagram of the 16-bit PWM mode.*

### *6.4.1 PWM DAC*

The PWM signal can also be used to generate analogue voltages if the signal is filtered with a low-pass filter. This way, a digital-to-analogue converter can be emulated. A simple first-order filtering is shown in Figure 6.12. The ripple of the signal depends on the filter and on the frequency of the PWN signal. If the ripple allowed at the PWM frequency is given, the filter corner frequency $1/(2\pi RC)$ can be determined.

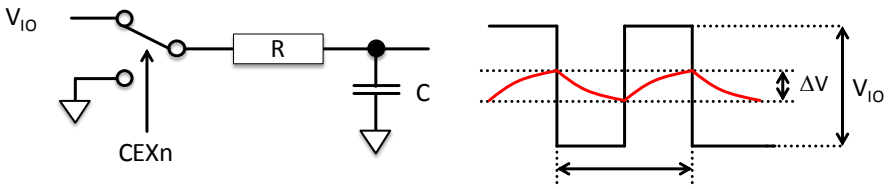


*Figure 6.12. The PWM signal can be low-pass filtered to approximate a DC voltage with a low-ripple signal.*

A simple estimation can be made assuming a 50% duty cycle, which is the worst case. If $\Delta V$ is small, the capacitor charging current is nearly constant; therefore, $\Delta V$ can be approximated by the following formula

$$\Delta V \approx \frac{I \cdot \frac{T}{2}}{C} = \frac{\frac{V_{IO}}{2}}{RC}\frac{T}{2} \tag{6.2}$$

$$\frac{\Delta V}{V_{IO}} \approx \frac{T}{4RC}; \tag{6.3}$$

therefore, choosing

$$RC > \frac{T}{4}\frac{V_{IO}}{\Delta V} \tag{6.4}$$

will keep the ripple under the desired limit.

Note that the precision of the output signal is limited by the precision of the $V_{IO}$ supply voltage. The supply voltage tolerance is not strict; 10% is typical. If higher accuracy is required, external circuitry should be used.

## 6.5 Application guidelines

- The input clock must be configured first. Choose a frequency value that allows the desired timing to be accurately set. If the clock frequency is too high, longer timing can be impossible. If it is too low, the accuracy of the timing can be low.
- Verify the settings by calculating the timing using the PCA SFR values.
- Configure the compare/capture modules in the required mode.
- After proper setup, enable the PCA counter and the modules used.
- Enable the PCA interrupt if needed. Do not forget to clear the interrupt pending flag in the service routine. Note that all modules generate the same interrupt; therefore, all possible requests must be handled within a single service routine and the corresponding pending flag must be cleared.
- Keep in mind that if the watchdog timer module is used, the PCA input clock cannot be changed while the watchdog timer is enabled.
- The 16-bit PCA counter value can be safely read by reading the lower-order byte (**PCA0L**) first.
- Always write the lower-order byte of the compare/capture registers (**PCA0CPLn**) first. Even if the higher-order byte is not changed, it must be written to re-enable the PCA comparator, which is disabled by writing to the lower-order byte. If only the higher-order byte needs updating, writing to the lower-order byte is not required.

## 6.6 Troubleshooting

**Problem:**

- The PCA is not running or unexpected timing occurs.

**Possible reasons:**

- The PCA is not enabled.
- The PCA modules are not configured in the proper mode.
- The input clock is not configured properly.
- The input clock source is missing – for example, the external signal is missing or Timer 0 overflows do not occur.
- The SRF values are miscalculated or not properly written.
- The higher-order byte of the capture/compare register is not written, so the PCA comparator can permanently remain in a disabled state.

**Problem:**

- No PCA interrupt occurs or the interrupt rate is not as expected.

**Possible reasons:**

- The PCA or the modules are not enabled.
- The associated interrupt is not enabled.
- The interrupt flag is not cleared; therefore, the interrupt is generated continuously.
- Execution of other interrupt service routines can cause delay of the timer interrupt.
- The service routine may take longer than the time between two PCA interrupt requests; the interrupt request rate is too high.

**Problem:**

- Unexpected frequency, period or pulse width of an external signal is experienced.

**Possible reasons:**

- The crossbar is not configured properly to connect the external signal to the PCA.
- The PCA settings (such as input clock or mode) are improperly set or miscalculated.
- The resolution of the time measurement is too low; for example, short periods are measured by only a few timer increments.
- The external signal is noisy, and oscillations occur at transitions.
- The PCA interrupt requests are generated faster than the interrupt service routine can handle them.

## 6.7 Exercises

- *Write code that can measure the width of a button pressing pulse using the edge-triggered capture mode. Solve the problem both with polling and with interrupt techniques*
- *Write code that generates a signal that is toggled with the following timing: 1 ms, 2 ms, 4 ms and 8 ms, and repeats this sequence infinitely. Use the high-speed output mode to implement the code. Check the result on an oscilloscope.*

- *Write a program that drives an LED with a PWM signal at a rate of 1000 Hz, with the pulse width set by button pressings from 10% to 90% by steps of 10% in a cyclic manner.*
- *Generate a 1-kHz PWM signal and pass it through a simple RC filter that reduces the ripple to 1%. Check the result with an oscilloscope.*

# 7 Serial communication peripherals

Today's electronic equipment is optimised for small size, low cost and reliability. Board space must be kept small and the wiring of the printed circuit board must be simple. Integrated circuits can be smaller if their pin count is small. Reliability is also improved with a lower number of contacts and a simpler design.

Since microcontrollers often communicate with other components, the above-mentioned requirements can be supported by serial interfaces that use only a few pins and wires to connect the devices. Microcontrollers provide several kinds of serial ports where the idea is to exchange bytes as bit streams, with one bit transferred at a time.

## 7.1 UART

One of the most popular serial interfaces is the so-called universal asynchronous receiver/transmitter (UART), developed with the aim of communicating with distant devices, using circuits that are typically on a separated printed circuit board [6]. Depending on the distance, a longer cable may be used to connect the devices, in which case a driver/receiver – aka transceiver – circuit is needed (for example, RS232 and RS485 transceivers). The data are sent over a single wire in one direction. The communicating devices have a transmit output (TX) and a receiver input (RX) pin. These must be cross-connected, i.e. the TX pin of one device should be connected to the RX pin of the other and vice versa. The interface is symmetrical: any side can send data asynchronously. If the TX pin can be disabled, even a single wire can be used for bidirectional data transfer. Sometimes one-directional data transfer is sufficient. Figure 7.1 summarises the connection possibilities.

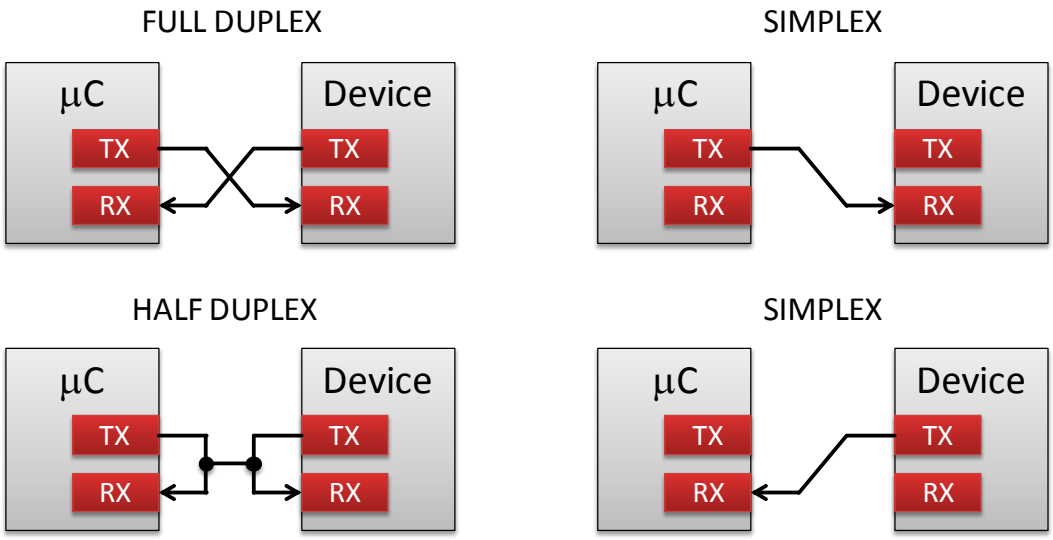


*Figure 7.1. Typical UART interconnections.*

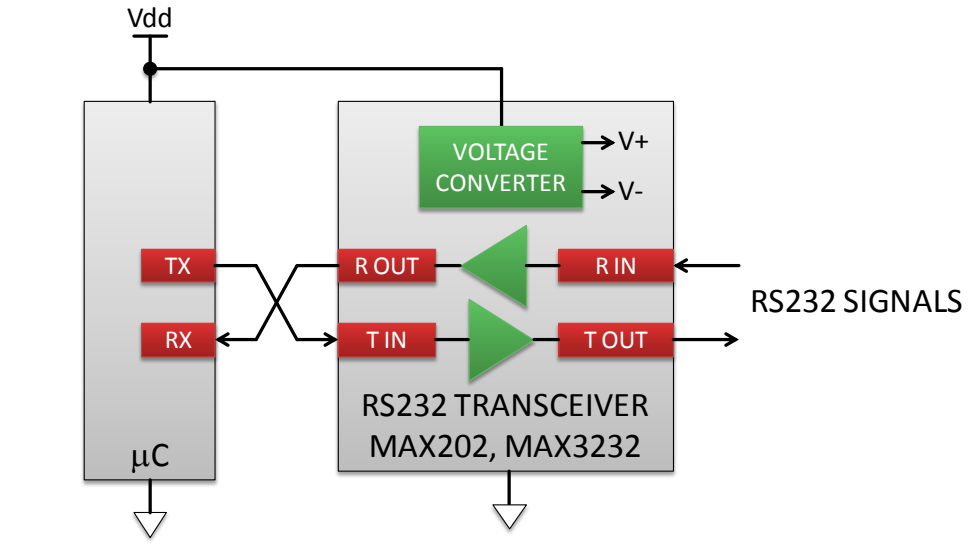


*Figure 7.2. RS-232 transceivers allow the use of longer cables between the communicating devices.*

The idle state of the signal is logic high; the transmission is started by setting the signal low for a given amount of time $\Delta t$. After sending this 'start bit', the data bits will be sent and each bit will be placed on the wire for time $\Delta t$. The transfer is terminated by a so-called stop bit, which is logic high for a duration of at least $\Delta t$. The transmitter and receiver must have the same timing; they detect the start bit and then sample the signal to determine the value of the bits. Sometimes a ninth bit is sent, which can be a parity bit or can be used in multiprocessor communication to mark the byte as a control or address word rather than data. Figure 7.3 shows the time diagram of the data transfer.

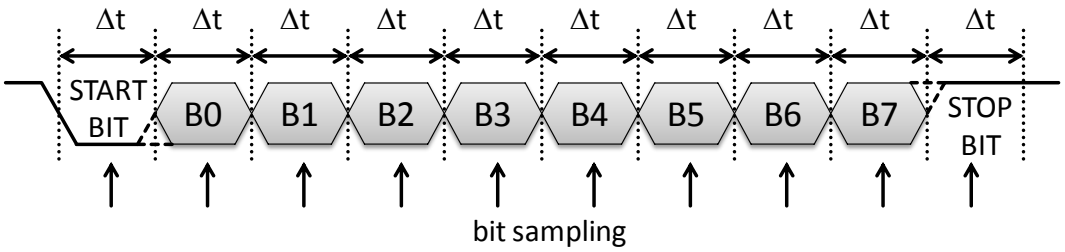


*Figure 7.3. Time diagram of the data transfer.*

The $1/\Delta t$ bit rate (called the *baud rate*) is generated by timer overflows in two different ways depending on which processor is used:

- baud rate = timer overflow rate / 16, (for example, C8051F120 UART 0)

or

- baud rate = Timer 1 overflow rate / 2 (for example, C8051F410 or C8051F120 UART 1).

Using Timer 1 in auto reload mode, the timer reload value can be determined in the following way:

- **TH1**=256-$T_{CLK0}$/(16·baud rate) (for example, C8051F120 UART 0),

or

- **TH1**=256-$T_{CLKo}$/(2·baud rate) (for example, C8051F410 or C8051F120 UART1).

If one of Timers 2–4 is used, the 16-bit reload value is

- **TMRRL**=65536- $T_{CLK}$/(16·baud rate).

The timers must be configured in auto reload mode and must be enabled. The associated timer interrupts are not enabled.

Note that the transmitter and receiver baud rate cannot be exactly the same since they are derived from different oscillators. Figure 7.4 shows a time diagram example of a 3% mismatch.

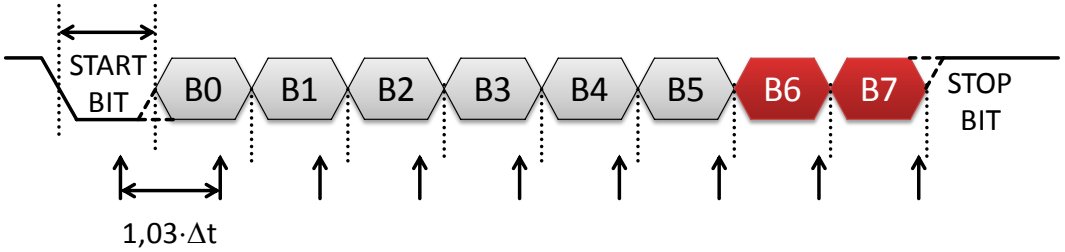


*Figure 7.4. Time diagram of data transfer with 3% baud rate tolerance. The red bits are not sampled properly.*

Depending on the signal transition time, the allowed tolerance is different. A higher baud rate or a longer transition time needs more strict matching [18]. It is strongly recommended to configure outputs as push-pull to keep transition times as short as possible.

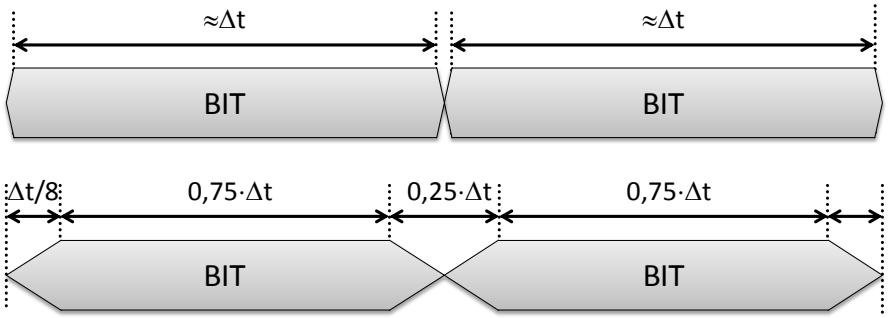


*Figure 7.5. Depending on the rise and fall time of the signals, the valid state can be longer, which allows a less strict tolerance of the baud rate.*

The following C8051F410 code examples illustrate simple polling-mode UART communication.

```
/*************************************************************************
UART initialisation function
*************************************************************************/
void UART_Init()
{
    SCON0 = 0x10; // 8-bit, variable baud mode
    TI=1;         // assume empty output buffer!
}
/*************************************************************************
UART input function, polling mode
```

```
*****************************************************************************/
unsigned char UARTIn(void)
{
    while (!RI); // wait for a byte
    RI=0;        // clear UART receive flag
    return SBUF; // return the byte
}
/*****************************************************************************
UART output function, polling mode
*****************************************************************************/
void UARTOut(char a)
{
    while (!TI); // wait for end of previous transmission
    TI=0;        // clear UART transmit flag
    SBUF=a;      // transmit a byte, do not wait for end
                 // this will also trigger the transmission process
}
```

The next example shows the use of ring buffers to transmit and receive in interrupt mode.

```
#define BUFFERSIZE 8

// declare ring buffers for input and output queue
volatile unsigned char TxBuffer[BUFFERSIZE];
volatile unsigned char RxBuffer[BUFFERSIZE];

// TX buffer read and RX buffer write pointers
// used in the interrupt routine
volatile unsigned char TxReadPtr=0, RxWritePtr=0;

// TX buffer write and RX buffer read pointers
unsigned char TxWritePtr=0, RxReadPtr=0;

// Number of data available in the ring buffers
volatile unsigned char TxNumberOfData=0;
volatile unsigned char RxNumberOfData=0;

/*****************************************************************************
UART interrupt routine
*****************************************************************************/
void UARTInterrupt(void) __interrupt UART_VECTOR
{
    if (RI)   // if byte has been received
    {
        RI=0; // clear UART receive flag
        if (RxNumberOfData < BUFFERSIZE) // does it fit in the buffer
        {
            RxBuffer[RxWritePtr]=SBUF;   // save the byte into the buffer
            RxWritePtr = (RxWritePtr+1) % BUFFERSIZE; // ring buffer indexing
            RxNumberOfData++; // increment number of received bytes
        }
    }
    if (TI)   // if byte has been transmitted
    {
        TI=0; // clear UART transmit flag
        if (TxNumberOfData) // if there are still bytes to be sent
        {
            // this will also trigger the transmission process
            SBUF=TxBuffer[TxReadPtr]; // send the byte
            TxReadPtr = (TxReadPtr+1) % BUFFERSIZE; // ring buffer indexing
```

```
            TxNumberOfData--; // decrement the number of bytes in the queue
        }
    }
}

/*************************************************************************
UART input function, interrupt mode
*************************************************************************/
unsigned char UARTIn(unsigned char *c)
{
    if (RxNumberOfData) // if bytes are available
    {
        RxNumberOfData--;  // decrement the number of available bytes
        *c=RxBuffer[RxReadPtr]; // read a byte from the buffer and return it
        RxReadPtr = (RxReadPtr+1) % BUFFERSIZE; // ring buffer indexing
        return 0; // return 0 if successful
    }
    return 1; // no byte could be read from the buffer
}

/*************************************************************************
UART output function, interrupt mode
*************************************************************************/
unsigned char UARTOut(unsigned char c)
{
    if (TxNumberOfData < BUFFERSIZE) // is there space in the transmit queue
    {
        TxNumberOfData++;  // increment number of bytes in the transmit queue
        TxBuffer[TxWritePtr]=c; // put the byte in the transmit queue
        TxWritePtr = (TxWritePtr+1) % BUFFERSIZE; // ring buffer indexing
        return 0; // return 0 if successful

    }
    return 1; // no byte could be placed into the transmit queue

}
```

### *7.1.1 Application guidelines*

- UART must be enabled on the crossbar and the TX output must be configured as push-pull.
- The baud rate should be set by configuring the associated timer overflow rate. The maximum baud rate is SYSCLK/16; however, for transmission, the baud rate can be SYSCLK/2 if the baud rate is equal to the timer overflow rate divided by 2.
- The timer must be enabled but the timer interrupt must not.
- The UART reception must be enabled.
- Either polling or interrupt mode can be used but the two modes should not be used simultaneously.
- If interrupt mode is used, the UART interrupt must be enabled. The interrupt pending flag must be cleared in the service routine. Note that both transmit and receive interrupts invoke the same service routine, so both events should be handled.
- The UART has a single-byte input buffer; therefore, the input data will be overwritten by the next incoming byte of data if the buffer has not been read by the processor in time.

- In order to avoid data loss, some kind of handshaking can be implemented. For example, a received byte can be sent back to confirm reception.

### *7.1.2 Troubleshooting*

**Problem:**

- The UART does not seem to send or receive data.

**Possible reasons:**

- The UART is not enabled on the crossbar or the crossbar is not enabled.
- The UART reception is not enabled.
- The UART baud rate timer is not enabled.
- The baud rate time is not configured properly.
- Broken or short-circuited wires or links between the communicating devices.

**Problem:**

- The data sent or received do not seem to be valid.

**Possible reasons:**

- The baud rates of the communicating devices do not match (due to improper settings or the limited accuracy of the internal oscillator, etc.).
- The baud rate is too high (higher than SYSCLK/16).
- The baud rate timer is used for another purpose and has been overwritten accidentally.
- The TX output signal is not configured as push-pull.

**Problem:**

- Some bytes are missing during data transfer.

**Possible reasons:**

- The receive buffer is not read in time by the processor before new data are received.
- The data transfer is too fast.

## *7.2 SPI*

Serial peripheral interface (SPI) is normally used to connect integrated circuits – sensors, ADCs, DACs, other microcontrollers, etc. – on the same board in a master-slave fashion [6]. SPI uses one wire for outgoing data (master out – slave in, MOSI) and another for incoming data (master in – slave out, MISO). A third wire (serial clock, SCK) driven by the master synchronises the transfer by providing a clock signal that changes when a bit of the data is available. Typically, the data are shifted out on one edge and can be read on the following opposite transition; the polarity can be chosen. An optional active low fourth signal (often called negated slave select, NSS) can be used to select the slave device, which ignores all communication signals if this line is inactive. This is useful to provide a safe frame (accidental pulses on the SCK line can corrupt data transmission) or to use multiple slave devices on the same bus.

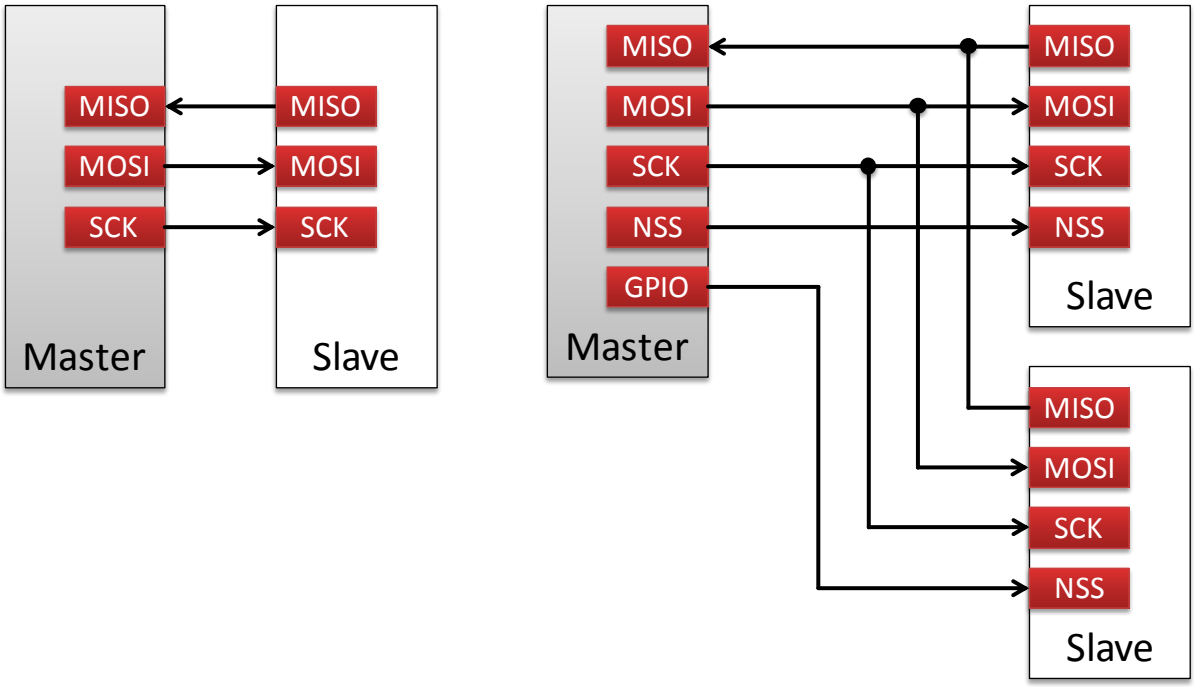


*Figure 7.6. SPI master and slave connections.*

The time diagram of a 3-wire transaction can be seen in Figure 7.7, while Figure 7.8 illustrates the use of the NSS signal. Note that there is no separate read operation; during a read the MOSI wire is driven. The slave device typically ignores this byte, but the datasheet must be consulted to ensure proper operation.

The SPI clock rate can be expressed as:

$$f_{SCK} = \frac{SYSCLK}{2 \cdot (SPI0CKR + 1)}, \tag{7.1}$$

where **SPI0CKR** is an SFR that can be given by the following formula:

$$SPI0CKR = \frac{SYSCLK}{2 \cdot f_{SCK}} - 1. \tag{7.2}$$

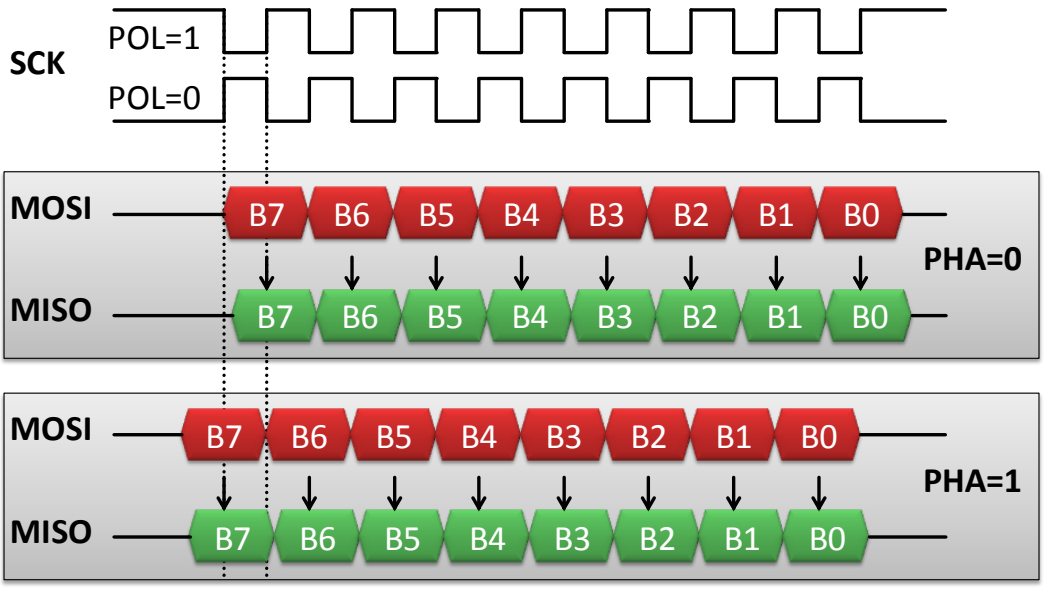


*Figure 7.7. Time diagram of an SPI transaction. The clock polarity and phase can be programmed.*

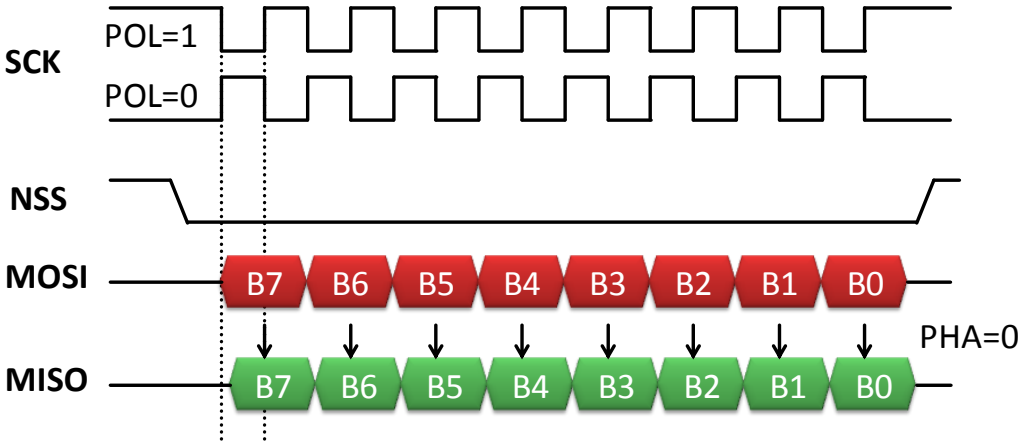


*Figure 7.8. Time diagram of a 4-wire SPI transaction.*

The following is a polling-mode SPI example code.

```
/*************************************************************************
SPI output function, polling mode
*************************************************************************/
void SPIOut(unsigned char c)
{
    SELECT  = 0;   // activate the select signal (negative logic)
    SPIF    = 0;   // clear the end of transmission flag
    SPI0DAT = c;   // place the byte into the transmit register
                   // this will also trigger the transmission process
    while (!SPIF); // wait until the end of transmission
    SELECT  = 1;   // deactivate the select signal (negative logic)
}

/*************************************************************************
SPI input function, polling mode
```

```
//-----------------------------------------------------------------------/
unsigned char SPIIn(void)
{
    SELECT  = 0;    // activate the select signal (negative logic)
    SPIF    = 0;    // clear the end of transmission flag
    SPI0DAT = 0;    // dummy write starts SPI clocking and therefore reception
    while (!SPIF);  // wait until the end of reception (8 bits)
    SELECT  = 1;    // deactivate the select signal (negative logic)
    return SPI0DAT; // return the received byte
}
```

### *7.2.1 Application guidelines*

- The SPI must be enabled on the crossbar and the outputs (MOSI and SCK in master mode, MISO in slave mode, and the select signal if applicable) must be configured as push-pull.
- The SCK clock rate should be set by setting the dedicated system clock divider value **SPI0CKR**. The maximum clock rate in master mode is the SYSCLK/2 or 12.5 MHz, whichever is lower; in slave mode, it must be less than SYSCLK/10.
- All parameters of the SPI port – clock phase, polarity, 3- or 4-wire mode and master or slave mode – must be set according to the communication requirements.
- Use either polling or interrupt mode, but not the two modes simultaneously.
- If interrupt mode is used, the SPI interrupt must be enabled. The interrupt pending flag must be cleared in the service routine. Note that several SPI events are handled by the same service routine, so care should be taken to handle interrupts from all possible sources in the routine.
- The SPI has a single-byte input buffer; therefore, the input data will be overwritten in slave mode by the next byte of incoming data if the buffer has not been read by the processor in time.

### *7.2.2 Troubleshooting*

**Problem:**

- The SPI does not seem to send or receive data.

**Possible reasons:**

- The SPI is not enabled on the crossbar or the crossbar is not enabled.
- The SPI is not configured properly.
- The select signal is used but inactive.
- Broken or short circuited wires or links between the communicating devices.

**Problem:**

- The data received or sent do not seem to be valid.

**Possible reasons:**

- The setup (clock phase, polarity, etc.) of the communicating devices does not match
- The clock rate is too high (higher than 12.5 MHZ or SYSCLK/10 in slave mode).
- The output signals are not configured as push-pull.
- The interface is not initialised properly; an accidental transaction has not finished.

## 7.3 SMBus

System management bus (SMBus) is a two-wire master-slave interface to connect multiple masters (microcontrollers) and multiple slaves (digital output sensors, ADCs, DACs, flash memory, etc.) on the same bus [6]. It is practically compatible with the I²C (Inter-integrated circuit) bus; a few minor differences include timeout handling, clock speed and line driving specifications. The communicating chips are typically found on the same printed circuit board or at least in the same equipment. The bus is not designed to use long wires (more than a few tens of centimetres).

One wire carries the data in both directions (serial data, SDA) and another (serial clock, SCL) is supplied by a master and synchronises the communication devices by clock pulses indicating valid bits on the bus.

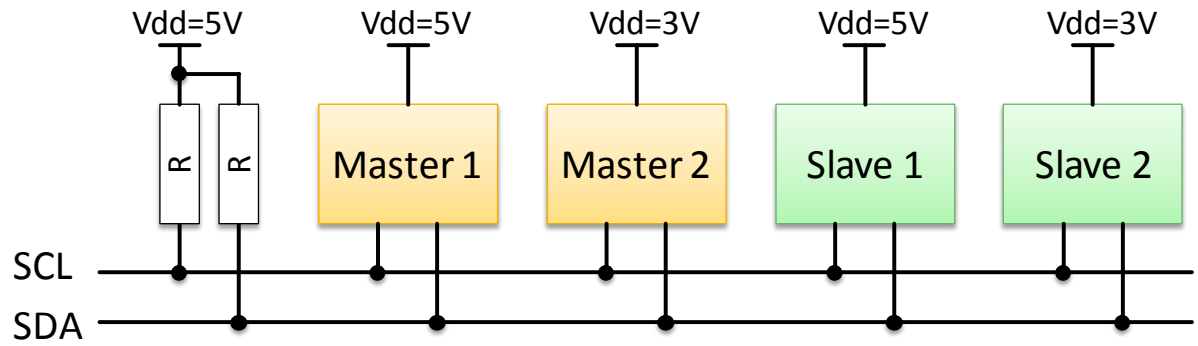


*Figure 7.9. Connecting devices to the two-wire SMBus.*

The SMBus has open-drain output drivers and needs external pull-up resistors for proper operation. Higher value resistors (about 10 kΩ or greater) lower the power consumption, while smaller resistances (down to about 2 kΩ) provide higher speed. The open-drain structure only allows both the master and the slave to set the signal logic low (pull-down); the resistors ensure logic high when none of the devices pull the signal down.

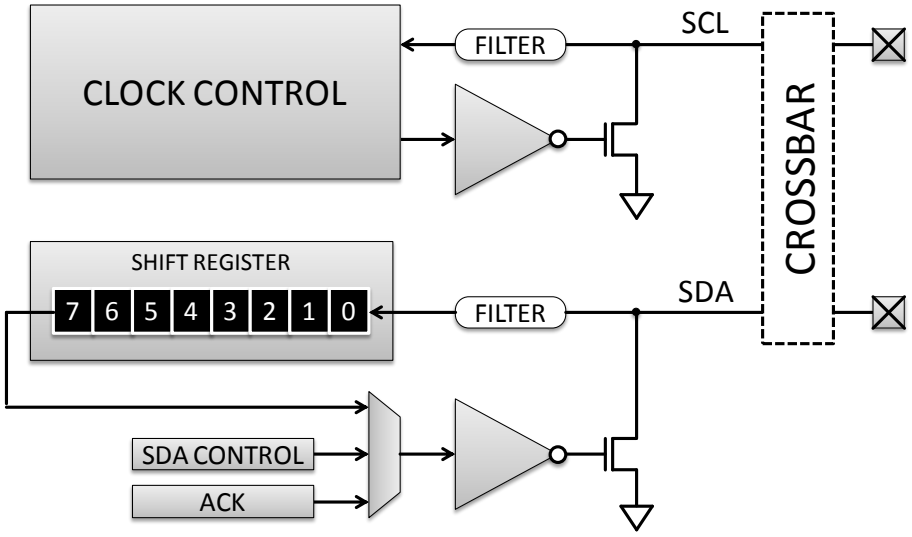


*Figure 7.10. Block diagram of the SMBus peripheral of the microcontroller.*

The bit rate can be set by Timer 0 and Timer 1 overflows and Timer 2 high or low byte overflows according to the following formula:

$$f_{SCL} = \frac{\text{Timer Overflow Rate}}{3}. \tag{7.3}$$

The time diagram of a typical transaction can be seen in Figure 7.11. In the inactive state both SDA and SCL are high. Data exchange is initiated by a start condition, which is the master pulling the SDA low. Then the master sends 7 address bits to select a device and a direction bit. This bit is logic high if the master reads from the slave and is logic low otherwise. After this, the slave must pull the SDA line low to acknowledge (ACK) the request; otherwise, the transaction will fail. Depending on the direction, the master or the slave then puts data on the SDA line, while the master controls the timing by driving the SCL line. Each byte must be acknowledged. After the last byte has been sent, either the master or the slave can send a not-acknowledge (NACK, release the SDA wire) to stop the data transfer. The transfer is ended by a stop condition: the master keeps SDA low while releasing the SCL to go high then releases the SDA line to let the external resistor to pull the line high.

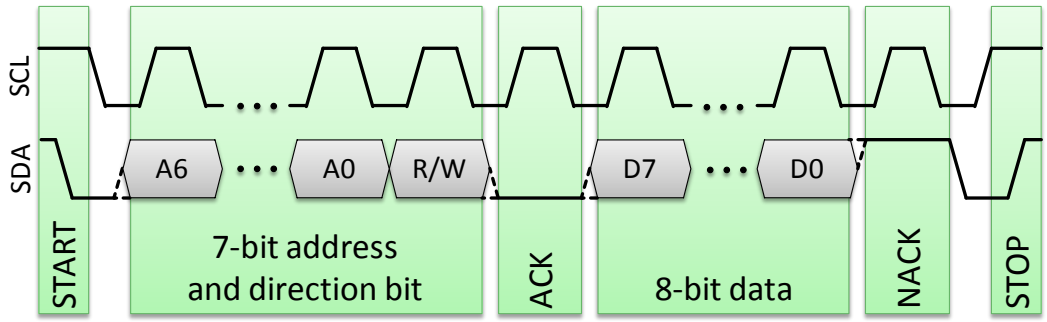


*Figure 7.11. Time diagram of an SMBus transaction.*

The communication is more complicated than in the case of UART or SPI. Two examples in Figure 7.12 and 7.13 show read and write time diagrams and interrupt flag behaviour. It is recommended to handle the transfer in an interrupt routine; however, polling the interrupt bit (SI) can be easier to implement and understand in simple transfers.

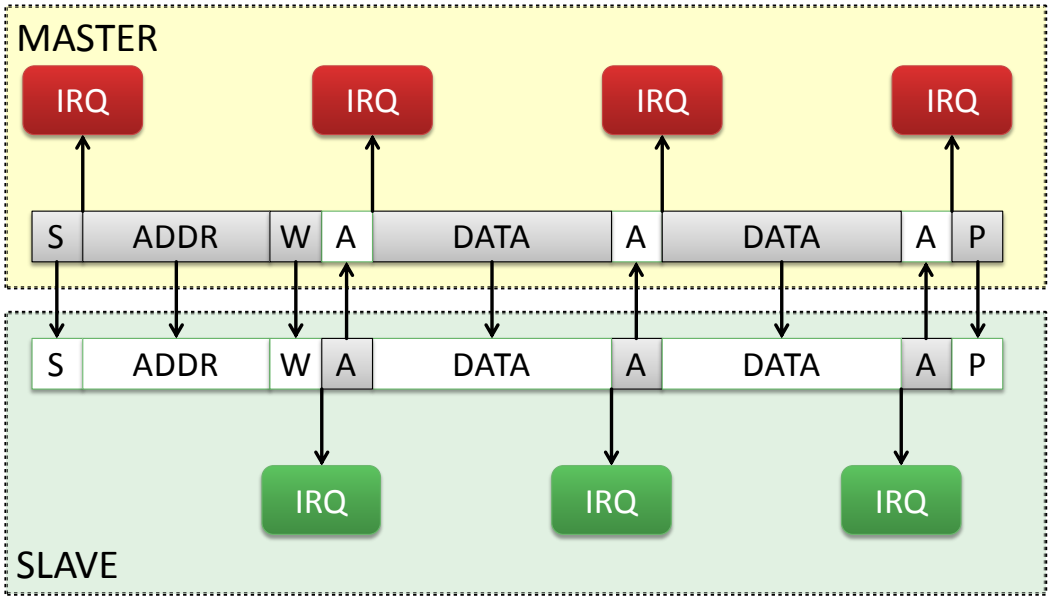


*Figure 7.12. SMBus write operation.*

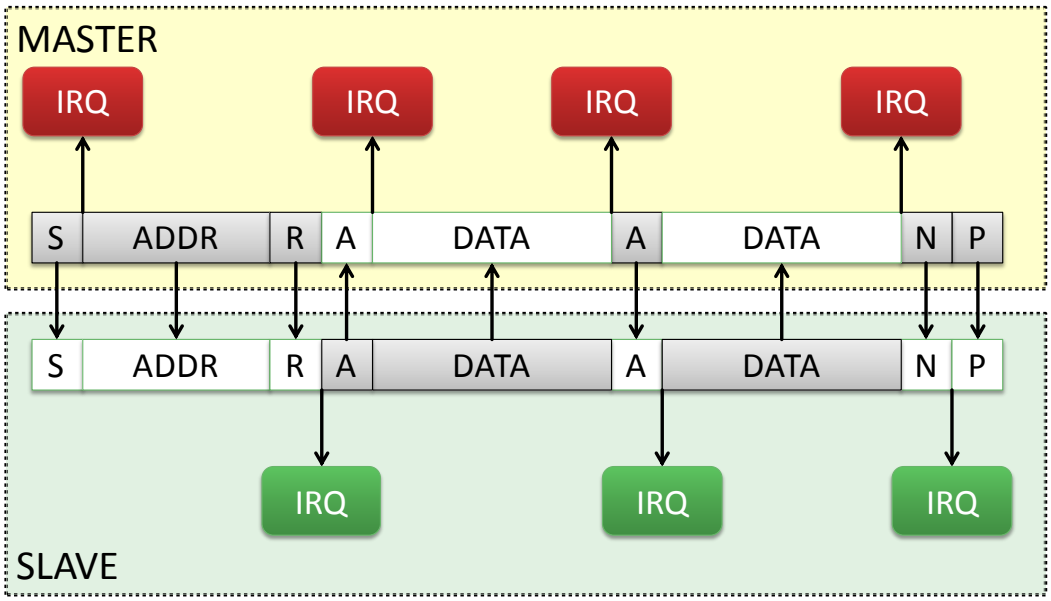


*Figure 7.13. SMBus read operation.*

A simple polling mode example code can be seen below.

```
/*************************************************************************
SMBus/I2C output function, polling mode
*************************************************************************/
void SMBusOut(unsigned char address, unsigned char c)
{
    STO = 0;                  // stop condition bit must be zero
    STA = 1;                  // start transfer
    SI = 0;                   // continue
    while (!SI);              // wait for start to complete
    STA = 0;                  // manually clear STA
    SMB0DAT = address << 1;   // A6..A0 + write
```

```
    SI = 0;                  // continue
    while (!SI);             // wait for completion
    if (!ACK)                // if not acknowledged, stop
    {
        STO = 1;             // stop condition bit
        SI = 0;              // generate stop condition
        return;
    }
    SMB0DAT = c;             // put data into shift register
    SI = 0;                  // continue
    while (!SI);             // wait for completion
    STO = 1;                 // stop condition bit
    SI = 0;                  // generate stop condition
}

/*************************************************************************
SMBus/I2C input function, polling mode
*************************************************************************/
unsigned char SMBusIn(unsigned char address)
{
    STO = 0;                 // stop condition bit must be zero
    STA = 1;                 // start transfer
    while (!SI);             // wait for start to complete
    STA = 0;                 // manually clear STA
    SMB0DAT = (address << 1) | 1;  // A6..A0 + read
    SI = 0;                  // continue
    while (!SI);             // wait for completion
    if (!ACK)                // if not acknowledged, stop
    {
        STO = 1;             // stop condition bit
        SI = 0;              // generate stop condition
        return;
    }
    ACK = 0;                 // NACK, last byte
    SI = 0;                  // continue
    while (!SI);             // wait for completion
    STO = 1;                 // stop condition bit
    SI = 0;                  // generate stop condition
    return SMB0DAT;
}
```

### *7.3.1 Application guidelines*

- The SMBus must be enabled on the crossbar and the associated two pins must be configured as open-drain.
- The SMBus data rate should be set by configuring the associated timer overflow rate. The maximum clock rate is SYSCLK/20. In most cases, rates close to the standard 100 kbit/s are used
- Master or slave mode can be selected.
- The associated timer must be enabled but the timer interrupt must not.
- Use either polling or interrupt mode, but not the two modes simultaneously.
- If interrupt mode is used, the SMBus interrupt must be enabled. The interrupt pending flag must be cleared in the service routine. Note that several SMBus events are handled by the same service routine, so care should be taken to handle interrupts from all possible sources in the routine..

- External pull-up resistors must be used. The typical range is from 2 kΩ to 10 kΩ. Lower values enable higher speed; higher values lower the power consumption.

### *7.3.2 Troubleshooting*

**Problem:**

- The SMBus does not send and receive data or the data sent or received do not seem to be valid.

**Possible reasons:**

- The SMBus is not enabled on the crossbar or the crossbar is not enabled.
- The SMBus is not configured properly.
- The SMBus data rate timer is not enabled.
- The data rate is not configured properly.
- The device address is invalid or no device is present on the bus.
- Broken or short-circuited wires or links between the communicating devices.
- The device communication protocol is not followed properly.
- The data rate (clock frequency) is too high and the clock pulse width is too narrow.
- The devices are too far from each other, and if they are connected with a cable, it may be too long.

## *7.4 C standard I/O redirection*

The standard C input/output functions in the SDCC environment use the putchar and *getchar* basic functions [4]. Therefore, it is possible to redirect the input and output to use a serial port. The *printf*, *scanf* and other functions will use the port. A code example of using the UART port can be seen below.

```
/*************************************************************************
Redirected standard C output function
*************************************************************************/
void putchar(char c)
{
    UARTOut(c);       // UART, SPI, SMBus, etc.
}

/*************************************************************************
Redirected standard C input function
*************************************************************************/
char getchar(void)
{
    return UARTIn(c); // UART, SPI, SMBus, etc.
}

…
printf("x=%d",x);     // example write operation
```

## 7.5 Exercises

- *Write code that sends back every received byte over the UART. The baud rate should be 9600 bit/s; use a personal computer connected to the UART via a USB-UART converter to the microcontroller board.*
- *Upgrade the UART example code to use the RTS and CTS handshake lines.*
- *Connect the MCP4141 digital potentiometer via the SPI port to the microcontroller. Write code to set the potentiometer value. Check the result with a digital multimeter.*
- *Connect the SST25VF020B 2-Mbit flash memory via the SPI port to the microcontroller. Write code to fill the first 256 locations with the location index and read back the data. Measure the SPI signals with an oscilloscope.*
- *Read the temperature value using an LM75 sensor via the SMBus interface. Display the value on the two 7-segment displays in degrees. Measure the SMBus signals with an oscilloscope.*

# 8 Analogue peripherals

Modern microcontrollers have several built-in analogue peripherals – comparators, ADCs, DACs and voltage references – to support very compact real-world signal processing. Some sensors can be connected even without additional analogue circuitry.

In this section, the analogue peripherals of the C8051F410 microcontroller will be detailed [6].

## 8.1 Comparators

Comparators are the simplest digitising components. They have a single logic output that is logic high if the voltage on the positive input is greater than the voltage connected to the negative input. Some hysteresis turns the comparator into a Schmitt trigger, making it less sensitive to the noise on slowly changing signals, which can cause oscillations on the output when the signal is close to the switching threshold.

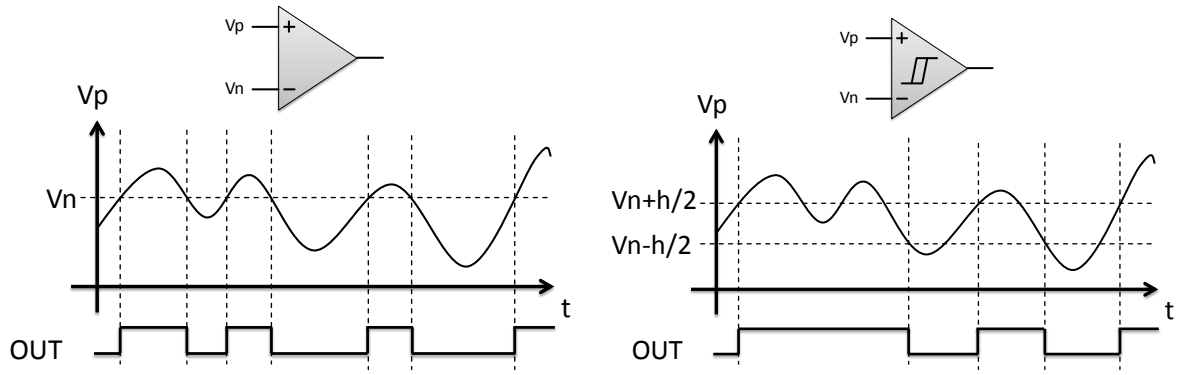


*Figure 8.1. Comparator and Schmitt trigger (comparator with hysteresis) waveforms.*

Note that comparators have some delay that varies from device to device and may also depend on the settings.

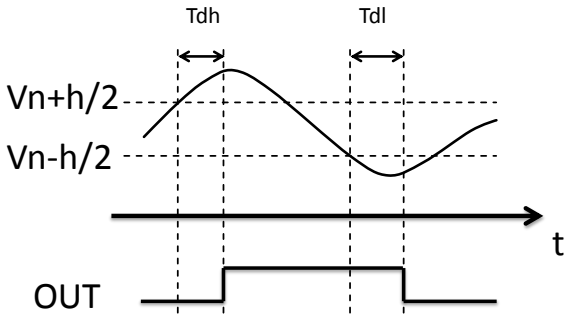


*Figure 8.2. Comparator delay.*

Comparators in C8051F410 processors have analogue multiplexers at their inputs so the signals can be chosen during program execution. The corresponding port pins *must be configured as analogue input* using the crossbar. The output of the comparators can be polled, used to generate an interrupt or connected to the pins of the microcontroller via the crossbar. In the latter case, the output can be either synchronised to the internal system clock or left as is (analogue mode).

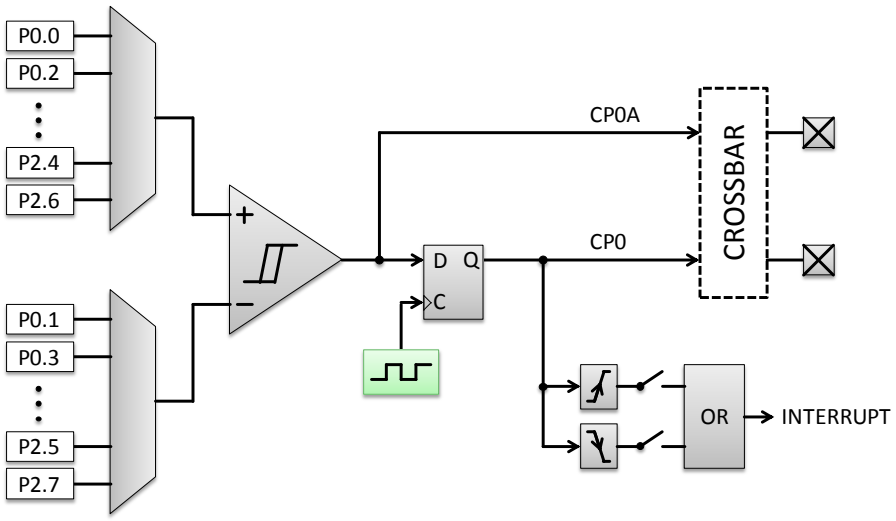


*Figure 8.3. C8051F410 comparator circuit.*

Comparators are useful for detecting levels of signals in applications such as heating control. They can be applied to convert a sinusoidal signal to a logic signal in order to measure its period; this way, the output of voltage-to-frequency converters can be digitised. They can also make the inputs compatible with different logic levels.

The comparator is useful for implementing a certain kind of analogue-to-digital converter. Figure 8.4 shows how it can be used to digitise the time constant *RC* of a resistor-capacitor circuit. When the push-pull output port bit is switched to logic high, the capacitor will be charged through *R* and the positive input is fixed at a fraction of the driving voltage. The time *t* needed to charge the capacitor to reach this voltage can be measured by a timer. This time can be obtained from the following derivation:

$$1-\exp\left(-\frac{t}{RC}\right)=\frac{R_1}{R_1+R_2}, \tag{8.1}$$

$$1-\frac{R_1}{R_1+R_2}=\exp\left(-\frac{t}{RC}\right), \tag{8.2}$$

$$\frac{R_1+R_2}{R_2}=\exp\left(\frac{t}{RC}\right), \tag{8.3}$$

$$t=RC\cdot\ln\left(\frac{R_1+R_2}{R_2}\right). \tag{8.4}$$

The logic high voltage level is not accurate, but the time does not depend on it, so the accuracy is not degraded. There are several resistive and capacitive sensors whose signals can be digitised using this method. Note that the input leakage current and the input capacitance can affect accuracy if the charging current is low or if the capacitor value *C* is low.

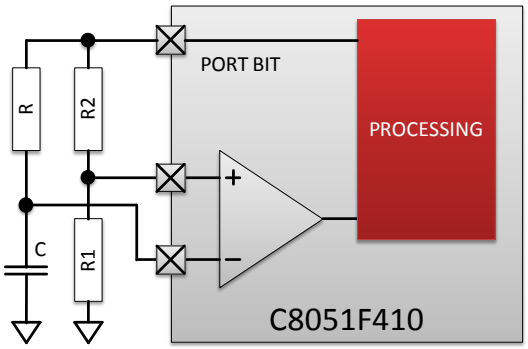


*Figure 8.4. The comparator can be used to measure the time constant of the RC circuit, thus R or C can be measured if one of them is known.*

### 8.1.1 Application guidelines

- The comparator input pins must be associated with port pins using the input multiplexer.
- The comparator input pins must be configured as analogue and must be skipped using the crossbar.
- The comparator input pins must not be left floating.
- The source impedance of the two input signals must be low and balanced to avoid an undesired voltage drop due to the input leakage current, and to reduce noise pickup.
- The comparator raw analogue output or the output synchronised to the system clock can be connected to a port pin using the crossbar.
- The comparator response time and power should be selected.
- The hysteresis should be configured. Noisier or slower signals typically need higher hysteresis.
- The input voltage range must be within the specifications; typically between ground and supply.
- The comparator interrupt can be generated on falling edge, on rising edge or on both transitions. If used, it must be enabled and the interrupt pending flag must be cleared in the interrupt service routine.

### 8.1.2 Troubleshooting

**Problem:**

- The comparator does not seem to detect the polarity of the voltage between its input terminals properly.

**Possible reasons:**

- The comparator is not enabled.
- The input multiplexer is not configured to assign the desired signals.
- The signals are out of the valid input range.
- The signals never cross each other; for example, one of the inputs is at GND while the other is always positive.
- The input signals are too fast; the polarity of the voltage between the input terminals is changing too quickly.
- An input is floating

- The source impedances of the two input signals are highly different; therefore, the input leakage current causes an undesired voltage drop.

**Problem:**

- The comparator interrupt is not generated.

**Possible reasons:**

- The comparator is not configured properly,
- The interrupt is not enabled or it is not configured properly.

**Problem:**

- The comparator output changes randomly or irregularly.

**Possible reasons:**

- The comparator is not configured properly.
- One or both of the inputs are floating or the source impedance is too high.
- The input signals are noisy or very slowly changing; the hysteresis is too small.

## *8.2 Voltage reference*

Most of the analogue peripherals need a stable, clean and accurate voltage. Comparators may need accurate voltage levels to which they can relate their input signals. Analogue-to-digital converters compare the input voltage to a reference voltage and determine the ratio of these voltages. Digital-to-analogue converters output a current or voltage that is an integer multiple of a small quantity. Although some use the supply voltage as a reference, it is absolutely not recommended, since the value is not stable enough: it depends on loading, it can be rather noisy and its accuracy is very poor. One must always use a precise dedicated reference voltage. Note that the accuracy of the reference determines the accuracy of all circuits that use it.

Mixed-signal microcontrollers have integrated voltage references but can use external references as well. Typically, the internal reference is not as accurate; some applications may need better performance. Typical parameters are the following.

**Absolute accuracy**. Guaranteed minimum and maximum voltage limits. Sometimes the error is given as percentage of the nominal voltage. Internal references have accuracies of 1%-2%, while external references can have accuracies below 0.1%.

The C8051F410 value is 1.8%.

**Temperature coefficient**. The reference voltage is somewhat temperature dependent. The value is given as ppm/K: parts-per-million change in the nominal voltage per degree. The value is typically around 30 ppm/K; for external references it can be below 3 ppm/K.

The C8051F410 value is 35 ppm/K.

**Load regulation**. The reference voltage depends slightly on the loading current. It can be given as voltage change per loading current or as ppm per loading current. A few ppm/μA is typical.

The C8051F410 value is 10 ppm/μA.

**Power supply rejection**. Changes in the value of supply voltage can cause small changes in the reference voltage. The smaller the change, the better the rejection. The ratio of the voltage reference change and the supply voltage change is given (mV/V or ppm/V).

The C8051F410 value is 2 mV/V.

**Maximum loading current**. Voltage references can drive external resistor dividers and other circuitry. However, the loading current cannot exceed a certain value; otherwise, the specifications are not guaranteed.

The C8051F410 value is 200 µA.

**Turn-on time**. The voltage reference can be turned on and off, which allows the use of an external reference and can help to optimise power management. Since the output voltage of the reference must be decoupled with capacitors (100 nF and 4.7 µF-10 µF), it takes a considerable amount of time – typically a few milliseconds – for the voltage to reach the final accurate value. It is a typical mistake to enable the reference and do an analogue-to-digital conversion without having waited for the reference settling time to pass. The error can be very large in this case.

The C8051F410 value is about 7 ms using 100 nF and 4.7 µF capacitors.

Figure 8.5 shows typical reference connections.

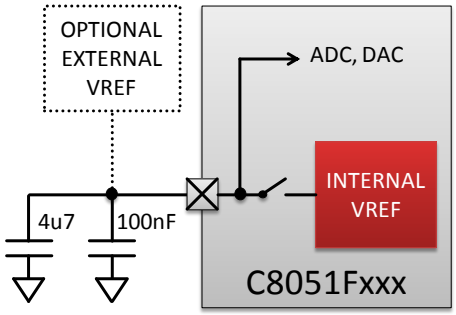


*Figure 8.5. Both external and internal reference can be used. The decoupling capacitors must be placed as close as possible to the reference pin.*

### 8.2.1 Application guidelines

- Decide whether to use internal or external reference. In both cases use decoupling capacitors.
- If internal reference is used, it must be enabled and the internal reference buffer should also be enabled.
- Do not overload the reference. It is a good practice to apply only loads well under the maximum (below 10%). Use an external operational amplifier buffer to provide higher current.
- Consider the reference turn-on time. Use it only after the value is surely stabilised.
- The reference port pin must be configured as analogue and must be skipped by the crossbar.

### 8.2.2 Troubleshooting

The troubleshooting for references is given in the following sections.

## 8.3 ADC

The analogue-to-digital converter outputs a $b$-bit binary number $d$ that depends on the input voltage as

$$d = \left\lfloor \frac{V}{V_{\text{ref}}} 2^b + 0.5 \right\rfloor, \tag{8.5}$$

where $V_{\text{ref}}$ is the voltage of the internal or external voltage reference [12]. The smallest voltage change that can be detected, $V_{\text{ref}}/2^b$, is called the voltage of the least significant bit ($V_{\text{LSB}}$) or voltage resolution. There are also differential ADCs, which measure voltage difference between their inputs and output a 2s complement binary number. Negative voltages are not allowed, because they are out of the range of the supply voltage — microcontrollers do not have a negative supply. This means that external signal conditioning is needed if the signal is out of range, if it is too small or if it is not a low-impedance voltage.

Figure 8.6 shows the block diagram of the 12-bit ADC integrated into the C8051F410 microcontroller. This successive approximation register (SAR) ADC is a very common architecture. Note that the conversion takes several steps (the number of bits plus 1), so a start signal and a periodic clock (SAR clock) signal are needed for proper operation.

The analogue multiplexer allows 27 different signals to be digitised. If the analogue signal (voltage) to be digitised is connected to a port pin, this pin must be configured as an analogue input using the crossbar and it must also be 'skipped', i.e. the crossbar must not assign any other peripherals to this pin. Note that the voltage reference pin (**P1.2**) must also be configured as an analogue input and skipped regardless of whether external or internal voltage reference is used.

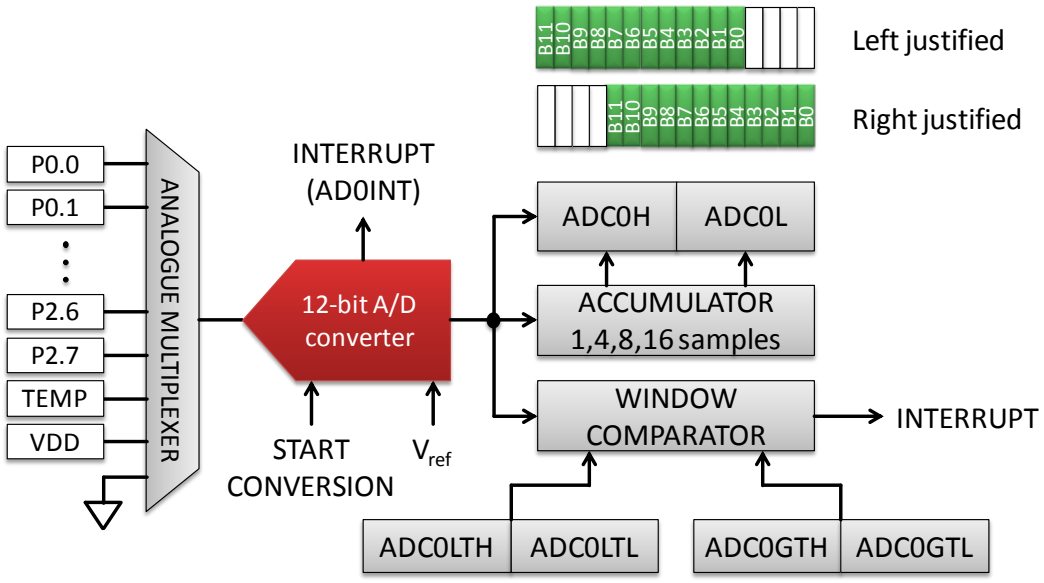


*Figure 8.6. The A/D converter circuit of the C8051F410.*

The 12-bit result of the conversion can immediately go to the 16 bits of SFRs **ADC0H** and **ADC0L** in left- or right-justified format. It is also possible to accumulate 1, 4, 8 or 16 samples and transfer their sum to the **ADC0H** and the **ADC0L**. In this case, the data must be right-justified, since the sum can take all 16 bits. A window comparator can set a flag or generate an interrupt depending on whether the result is in a specified range.

The conversion can be started in several ways, as shown in Figure 8.7. At the end of conversion the **AD0INT** flag is set and an interrupt can be generated.

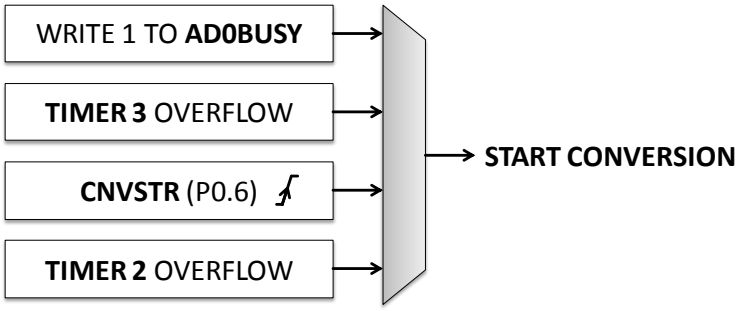


*Figure 8.7. Start of conversion sources.*

It is important to note that before conversion, the input signal is sampled by a capacitor $C_s$, as shown in Figure 8.8.

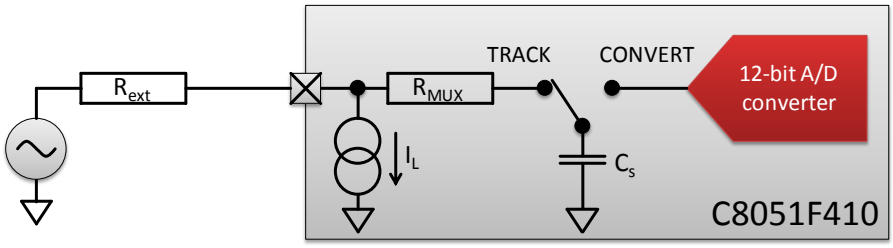


*Figure 8.8. Simplified schematic of the ADC input.*

The capacitor must be charged to a voltage that is close enough to the input voltage. If the deviation is less than half of the voltage resolution $\frac{1}{2} V_{LSB} = V_{ref}/2^{13}$, the error thus introduced does not degrade the 12-bit resolution. Since the capacitor is charged through $R_{ext}$ and the internal resistance $R_{MUX}$=5 kΩ, the sampling time must be at least

$$\left(R_{ext} + R_{MUX}\right) \cdot C_s \cdot \ln\left(2^{13}\right) = \left(R_{ext} + 5\,\text{k}\Omega\right) \cdot 12\text{pF} \cdot \ln\left(2^{13}\right) \approx \frac{R_{ext}}{\text{k}\Omega} 110\text{ ns} + 540\text{ ns}, \tag{8.6}$$

so at least a 540-ns signal tracking time is required. However, the datasheet specifies 1000 ns as minimum due to the uncertainty of the values 5 kΩ and 12 pF. In order to ensure reliable operation, it is best to *keep a minimum of 1000 ns tracking time and add 200 ns for every kΩ of output impedance of the signal source*:

$$t_{track} \geq \frac{R_{ext}}{1\,\text{k}\Omega} 200\text{ ns} + 1000\text{ ns}, \tag{8.7}$$

Even if a DC signal is measured, this minimum tracking (or sampling) time must be guaranteed, because the sampling capacitor is discharged during conversion.

Note that the analogue input has a leakage current. It flows through $R_{ext}$, therefore causes an error voltage. For higher impedances, an operational amplifier is recommended to provide low-impedance output.

A more common connection can be seen in Figure 8.9, when an external capacitor is placed between the input and ground. This capacitor can remove high-frequency noise, charge the sampling capacitor quickly and isolate the signal source from the current transients caused

by the switched sampling capacitor. The latter is especially useful when the signal source is the output of an operational amplifier. $C_{ext}$ is typically chosen to be much greater than the sampling capacitance. For example, if it is 1000 times greater, it can charge the sampling capacitor to 99.9% even without drawing current from the signal source. However, between conversions the external capacitor must be recharged to represent the input voltage with a specified accuracy.

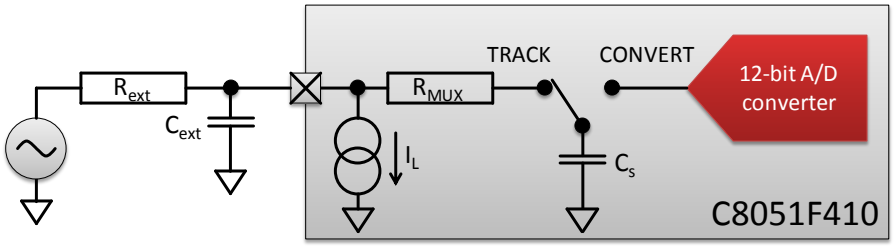


*Figure 8.9. Signal source connected to the input via a series resistor ($R_{ext}$) and a capacitor ($C_{ext}$).*

In order to make it clearer, an example follows. Let us assume that the voltage of the source is $V_{in}$ and the sampling frequency is $f_s$. This means that in every conversion cycle the sampling capacitor drains a charge of $V_{in} \cdot C_s$, so the average current flowing to the input is the charge divided by the sampling period $t_s$ (which is equal to $1/f_s$):

$$I = \frac{V_{in} \cdot C_s}{t_s} = V_{in} \cdot C_s \cdot f_s, \tag{8.8}$$

This current flows through $R_{ext}$, therefore causes an average voltage drop on it:

$$\Delta V = R_{ext} \cdot V_{in} \cdot C_s \cdot f_s; \tag{8.9}$$

therefore, the relative error can be estimated as

$$\frac{\Delta V}{V_{in}} = R_{ext} \cdot C_s \cdot f_s. \tag{8.10}$$

Thus at a given sample rate and relative error the $R_{ext}$ resistance is limited as

$$R_{ext} < \frac{\Delta V}{V_{in}} \frac{1}{C_s \cdot f_s}, \tag{8.11}$$

or for a given $R_{ext}$ value the sample rate must be limited as

$$f_s < \frac{\Delta V}{V_{in}} \frac{1}{C_s \cdot R_{ext}}. \tag{8.12}$$

For example, if 0.1% error is allowed, then at a sample rate of 10 kHz $R_{ext}$ must be less than $0.001/(12 \cdot 10^{-12} \cdot 10^4)\ \Omega \approx 8333\ \Omega$.

In summary:

- when no external capacitor is used, the minimum tracking time is given by Equation 8.7;
- using an external capacitor much (about 1000 times) greater than the sampling capacitor, the tracking time can be kept at its minimum, but the sample rate is limited according to Equation 8.12.

According to the above the external resistor and capacitor can only be used as an simple anti-aliasing filter if Equations 8.11 and 8.12 are satisfied. On the other hand, single pole filters do not reduce higher frequency components properly. If the signal contains significant components above $f_s/2$, then an active anti-aliasing filter is preferred. A popular simple second-order low-pass filter [11] is shown in Figure 8.10.

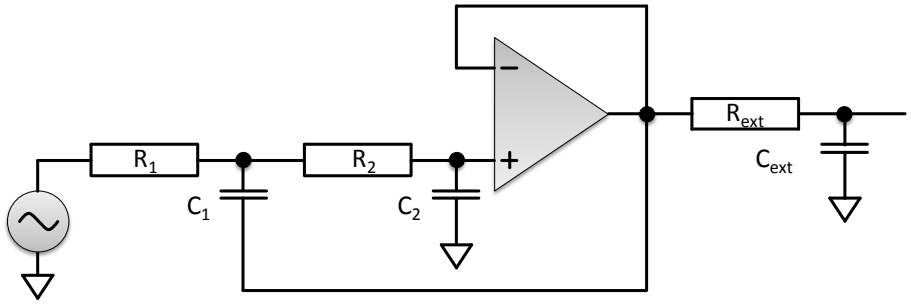


*Figure 8.10. Sallen-Key second-order low-pass filter. Note that $R_{ext}$ and $C_{ext}$ are not parts of the filter.*

The transfer function of this filter can be given as:

$$A(\omega) = \frac{1}{1 + \mathrm{i}\omega(R_1 + R_2)C_2 - \omega^2 R_1 R_2 C_1 C_2}, \tag{8.13}$$

where $\omega$ is the angular frequency: $\omega = 2\pi f$.

The general formula of a second-order low pass filter can be written as:

$$A(\omega) = \frac{1}{1 + \frac{\mathrm{i}\omega}{Q\omega_0} - \frac{\omega^2}{\omega_0^2}}. \tag{8.14}$$

The values of $Q$ and $\omega_0$ can be obtained from tables or by using filter design software, while the values of the resistors and capacitors can be determined.

Higher-order filters with better high-frequency rejection can be realised by cascading several first- or second-order stages. Note that $R_{ext}$ and $C_{ext}$ are not parts of the filter: these components are needed to isolate the output of the operational amplifier from the transient load caused by the switched sampling capacitor.

The C8051F410 microcontroller offers several tracking options that are illustrated in Figure 8.11.

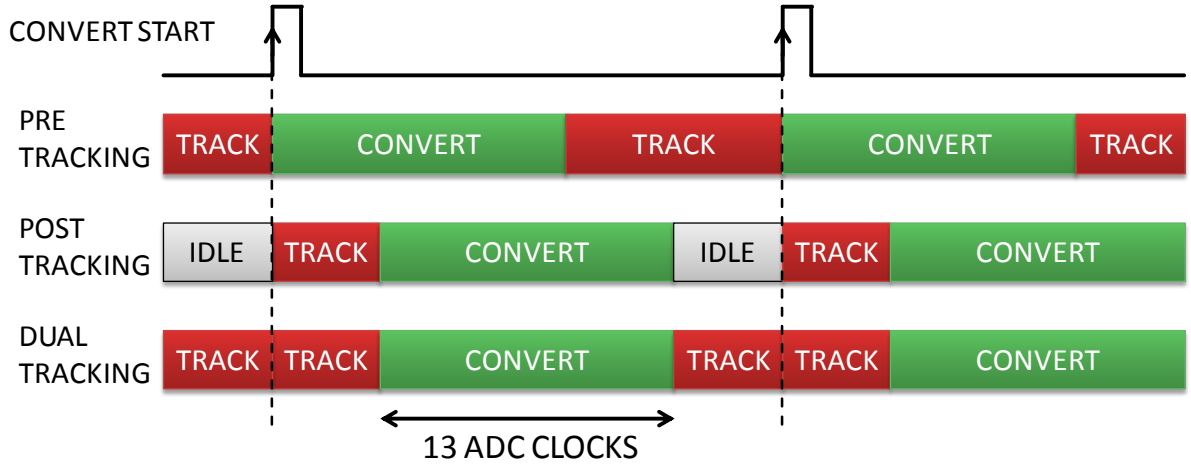


*Figure 8.11. Time diagram of the different tracking modes.*

The safest mode is the dual tracking mode. The post-tracking mode can be used to save power, since the ADC is in an idle state between conversions. The pre-tracking mode can help to achieve the highest possible conversion rate, but one must be very careful, because a minimum tracking time is not guaranteed. Therefore, the use of this mode is not recommended.

The following simple example code illustrates ADC handling in polling mode.

```
P0MDIN  = 0xFE;  // P0.0 analogue input
P1MDIN  = 0xFB;  // P1.2 analogue input (VREF)
P0SKIP  = 0x01;  // skip P0.0 since it is an analogue input
P1SKIP  = 0x04;  // skip P1.2 since it is an analogue input
REF0CN  = 0x13;  // enable internal VREF
ADC0CF  = 0x00;  // 191406-Hz ADC clock
ADC0CN  = 0x80;  // enable ADC (conversion start: set AD0BUSY)

unsigned int GetADC(unsigned char channel)
{
    ADC0MX = channel; // set the multiplexer
    ADC0CN = 0x80;    // enable the ADC
    AD0INT=0;         // clear the end of conversion flag
    AD0BUSY=1;        // start A/D conversion
    while (!AD0INT);  // wait for end of conversion
    AD0INT=0;         // clear the end of conversion flag
    return (ADC0H << 8)+ADC0L; // return the result of the A/D conversion
}
```

A more efficient way is to read the converted data in an interrupt service routine. One possible implementation can be seen below.

```
TMR2RLL = 0x60;  // high byte of reload register for a 100-Hz overflow rate
TMR2RLH = 0xFF;  // low byte of reload register for a 100-Hz overflow rate
TMR2CN  = 0x04;  // enable Timer 2
P0MDIN  = 0xFE;  // P0.0 analogue input
P1MDIN  = 0xFB;  // P1.2 analogue input (VREF)
P0SKIP  = 0x01;  // skip P0.0 (input signal)
P1SKIP  = 0x04;  // skip P1.2 (VREF)
REF0CN  = 0x13;  // enable internal VREF
ADC0CF  = 0x00;  // 191406-Hz ADC clock
ADC0CN  = 0x83;  // enable ADC (conversion: TIMER 2)
EIE1    = 0x08;  // enable ADC interrupt
```

```
IE      = 0x80;  // enable interrupts

/*************************************************************************
ADC interrupt handler routine
*************************************************************************/
void ADC_interrupt(void) __interrupt ADC_VECTOR
{
    AD0INT = 0; // clear the end of conversion flag
    adc_data = (ADC0H << 8) | ADC0L; // save the result of the A/D conversion
}
```

### *8.3.1 Application guidelines*

- The ADC should be enabled for proper operation.
- Select the event that starts the conversion: it can be the write signal to **AD0BUSY** SFR bit, the overflow of Timer 2 or Timer 3 or a rising edge of an external signal (CNVSTR, **P0.6**). If CNVSTR is used, the port pin must be skipped and configured as open-drain and 1 must be written to the corresponding port bit.
- If a timer overflow is used to start a conversion periodically, the timer must be configured properly, and the timer interrupt should not be enabled.
- Select the input signal by setting the multiplexer to the desired port pin. The pin must be configured as analogue and must be skipped using the crossbar. If multiple signals must be converted, then all associated pins must be configured as analogue and must be skipped.
- Select the desired ADC SAR conversion clock. Choose the highest frequency available but not higher than the specified maximum (3 MHz for the C8051F410). The full conversion takes 13 cycles plus the tracking time.
- If the system clock frequency is low or low-power operation is needed, the use of burst mode is recommended. In this mode, the ADC is operated from a high-speed clock independent of the system clock and is only out of idle state during conversion.
- Select the proper tracking mode and post-tracking time. Dual tracking mode is preferred in most cases, since it guarantees a minimum tracking time. Post-tracking mode can be used when low-power operation is needed.
- Consider the output impedance $R_{ext}$ of the signal, the external capacitance $C_{ext}$, the internal resistance of the multiplexer and the value of the sampling capacitor to estimate the minimum tracking time and maximum sample rate using Equations 8.7 and 8.11–8.12. The input leakage current flows through $R_{ext}$, so it also causes an error. An operational amplifier can be used if the impedance is high.
- The voltage reference pin (**P1.2**) must be configured as analogue and must be skipped using the crossbar. If the internal voltage reference is used, it must be enabled and the internal reference buffer must be enabled. The internal bias generator must also be enabled.
- If the conversion is started by writing to the **AD0BUSY** bit, polling the **AD0INT** bit can be used to wait for the end of conversion. The **AD0INT** bit must be cleared before starting the conversion. In multichannel applications, the next channel must be selected just after the end of conversion.
- If the conversion is started by a timer overflow or by an external signal (CNVSTR), then the end of conversion event should be handled by the ADC interrupt service routine. In the routine, the **AD0INT** flag must be cleared and in multichannel

applications, the next channel must be selected at the beginning of the routine to allow the longest possible settling time.
- The 12-bit ADC can be left- or right-justified. Multiple (4, 8 or 16) samples can be accumulated and then their sum can be read. In this mode, the data must be right-justified, because addition would cause an overflow otherwise. If 16 samples are accumulated, the result will be a 16-bit word. Note that averaging may reduce the noise in certain cases but does not improve accuracy. Taking 16 samples can reduce the noise to one fourth.

### *8.3.2 Troubleshooting*

**Problem:**

- No A/D conversions can be detected.

**Possible reasons:**

- The ADC is not enabled.
- Interrupt mode is planned but the ADC interrupt is not enabled or the global interrupt flag is disabled.
- The start of conversion signal is missing or not configured properly. If timers are used, they might not be enabled. The external start of conversion signal pulse can be too narrow.

**Problem:**

- The conversion result is not valid.

**Possible reasons:**

- The port pin is not configured as an analogue input.
- Due to improper multiplexer settings, the signal is not connected to the ADC input.
- The voltage reference or the internal bias generator is not enabled.
- Internal reference is used, but the internal reference buffer is not enabled.
- The voltage reference is enabled just before starting a conversion. Note that the voltage reference stabilisation time can be several milliseconds, which must be allowed to pass before starting a conversion.
- The voltage reference is overloaded, so it does not provide the proper value.
- Polling mode is used and the data are read before the end of conversion. The **AD0INT** flag might not be logic low before starting a conversion.
- Improper integer data handling occurred. For example, left-justified or accumulated data must be stored in an unsigned short.
- The ADC SAR clock frequency is too high (>3 MHz) or too low.
- The tracking time is too short. The signal output impedance might be too high, which necessitates a longer tracking time; see Equation 8.7.
- The signal output impedance is high, so the input leakage current causes significant error.
- The signal is out of the measurement range (0–$V_{ref}$).

## 8.4 DAC

The rich set of analogue peripherals of the C8051F410 includes two independent 12-bit current output digital-to-analogue converters. The output current range (full-scale output current, $I_{max}$) can be set as 0.25 mA, 0.5 mA, 1 mA or 2 mA.

Current output DACs are typically faster, but need external circuitry if voltage output is needed. Even a simple resistor of value $R$ suffices (see Figure 8.12), but the specified compliance range must be met, and the output voltage must be below $V_{dd}$-1.2 V. Note also that in such a configuration, the output impedance is $R$. For example, 1 V output range at 1 mA full-scale current requires $R$=1 kΩ.

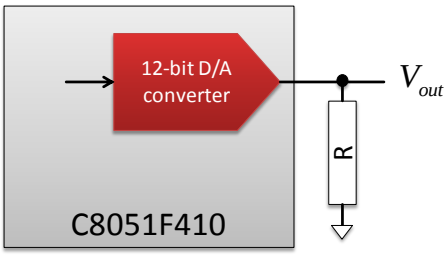


*Figure 8.12. A resistor converts current to voltage for the C8051F410 current output DACs. The output voltage is $V_{out}=R \cdot I_{max} \cdot N/2^{12}$.*

The digital-to-analogue conversion can be initiated by simply writing to the DACs SRFs (**IDA0L** first then **IDA0H**), but for the most accurate timing – needed for example for waveform generation –, timer overflow or the transitions of an external signal can be used, as shown in Figure 8.13.

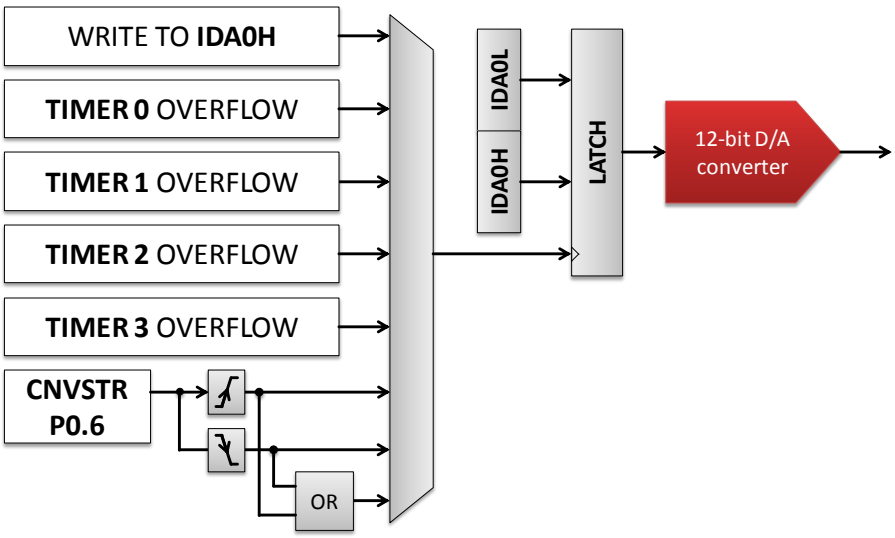


*Figure 8.13. Sources that can control the DAC output update.*

A simple example of using DAC 0 with output update upon writing to IDA0H:

```
P0MDIN = 0xFE; // P0.0 analogue mode
P0SKIP = 0x01; // skip P0.0
IDA0CN = 0xF2; // enable DAC0, update by write to IDA0H
               // 1 mA full scale, left-justified data
```

```
IDA0L = 0;     // low byte of the DAC output register
// writing to the high byte of the DAC output register updates the DAC output:
IDA0H = 128;   // high byte of the DAC output register, half scale: 0.5mA
```

The following code updates the DAC 0 output at a rate of 100 Hz and generates a ramp signal:

```
unsigned short dac_data = 0;
P0MDIN = 0xFE;   // P0.0 analogue mode
P0SKIP = 0x01;   // skip P0.0
IDA0CN = 0xA2;   // enable DAC0, update: Timer 2 overflow
                 // 1 mA full scale, left justified data
TMR2RLL = 0x60;  // high byte of the reload register for 100-Hz overflow rate
TMR2RLH = 0xFF;  // low byte of the reload register for 100-Hz overflow rate
TMR2CN  = 0x04;  // enable Timer 2

IE      = 0xA0;  // enable Timer 2 a global interrupts

/***********************************************************************
Timer 2 interrupt handler routine
***********************************************************************/
void Timer2_interrupt(void) interrupt TIMER2_VECTOR
{
    TF2H=0;               // clear interrupt pending flag
    IDA0L=dac_data;       // set lower order byte
    IDA0H=dac_data >> 8;  // set higher order byte
    dac_data++;           // increment the value for linear ramp
}
```

### *8.4.1 Application guidelines*

- The port pin associated with the DAC output should be set as analogue and must be skipped using the crossbar.
- Select the DAC output current range.
- Select the DAC output update source: write to the DAC register, timer overflow, or external signal (CNVSTR, **P0.6**). If CNVSTR is used, then it must be configured as open-drain and must be skipped. If the update source is timer overflow or an external signal, an associated interrupt service routine should set the next DAC value.
- The DAC update can be precisely synchronised to ADC conversions. In this case, the ADC interrupt routine must update the DAC registers and the ADC start of conversion and the DAC update source must be the same (timer overflow or CNVSTR).
- Enable the DAC and the internal bias generator.
- If voltage output is needed, connect a resistor between the output and ground. The output compliance range ($V_{dd}$–1.2 V) should not be exceeded, so at full-scale current the voltage on the resistor must be within this range.
- Select left or right justification. If only the 8 most significant bits are used, left-justified mode should be selected, since only the higher-order byte is used in this case.
- If used, the lower-order byte (**IDA0L** or **IDA1L**) must be written first. The DAC output is updated if the higher order byte is written (**IDA0H** or **IDA1H**), unless the update method is configured differently.

### *8.4.2 Troubleshooting*

**Problem:**

- The DAC output is unchanged or invalid when writing to the DAC registers

**Possible reasons:**

- The DAC or the bias generator is not enabled.
- Only the lower-order byte (**IDA0L**) is written.
- The output update source is a timer, but it is not configured properly or it is not enabled.
- If an interrupt routine must update the DAC output, the interrupt might not be enabled.
- The output is not configured as analogue or is not skipped. The port pin is used by another peripheral or port latch.
- The output compliance range is violated.
- The output update rate is too high, whereas the DAC output needs a certain settling time.

## *8.5 Temperature sensor*

The C8051F410 microcontroller has an internal diode-based temperature sensor, whose output voltage can be measured by the ADC. The voltage depends almost linearly on the chip temperature:

$$V = a \cdot T + b, \tag{8.15}$$

where the value of $a$ is 2.950 mV/°C±0.073 mV/°C and $b$=900 mV±17 mV.

The linearity error – the maximum deviation from the ideal linear dependence – is ±0.2 °C; the overall accuracy is about ±3 °C.

Note that depending on the operating power the chip temperature can be significantly higher than the ambient temperature. In a low-power application (below 5 mW), the chip temperature is close to the temperature of the printed circuit board (approximately within 1 °C–2 °C).

The temperature sensor can be used to monitor the operating temperature of the chip and the containing printed circuit board. In some very low power applications it can be used to estimate the ambient temperature with an accuracy of about ±3 °C. Temperature changes can be monitored more accurately.

## *8.6 Exercises*

- *Measure the resistance of a resistor in the range of 1 kΩ to 99 kΩ using the comparator as shown in Figure 8.4. Use C=100 nF, $R_1$=27 kΩ and $R_2$=47 kΩ. Display the result in kΩ units on two 7-segment displays. Measure 1% precision resistors (1 kΩ, 3.3 kΩ, 10 kΩ and 33 kΩ) and compare the results with the nominal values.*
- *Measure the capacitance of a capacitor in the range of 1 nF to 99 nF using the comparator as shown in Figure 8.4. Use R=100 kΩ, $R_1$=27 kΩ and $R_2$=47 kΩ. Display the result in nF units on two 7-segment displays. Measure 1 nF, 3.3 nF, 10 nF and 33 nF capacitors and compare the results with the nominal values. Also consider the precision of the nominal values.*
- *Try to measure the internal voltage reference turn-on time using the ADC.*

- *Read the position of the potentiometer with the ADC and display the value converted into 0 to 99 on two 7-segment displays.*
- *Write a program that generates a sinusoidal waveform using the DAC. One period should contain 64 points; the output data rate defined by a timer overflow should be 100 kHz; use a 24.5-MHz system clock. Check the result with an oscilloscope.*
- *Implement a voltage-to-frequency converter by measuring the voltage at the output of the potentiometer and generate a logic signal whose frequency is a linear function of this voltage. The frequency range should be 1 kHz to 10 kHz. Use the PCA frequency output mode.*
- *Write code that can delay a signal by 100 µs to 10000 µs in 100 µs units. Convert the signal at a rate of 10 kHz and output the delayed signal on DAC0. Use a timer to synchronise the ADC sample rate and DAC update rate. The delay value should be set by the potentiometer. Check the result on an oscilloscope.*
- *Measure the on-chip temperature using the internal temperature sensor. Measure the on-chip temperature as a function of the system clock frequency.*
- *Find a method to estimate how much higher the on-chip temperature is than the temperature of the printed circuit board.*

# 9 Sensor interfacing

There are many different sensors that can be interfaced to mixed-signal microcontrollers. In most cases, some external analogue signal conditioning circuitry is needed [11]. Since the microcontroller is a single-supply device, level shifting is used to handle bipolar signals. External active signal conditioning is typically based on single-supply operational amplifiers that may need additional attention.

In the following the most important solutions are presented briefly.

## 9.1 Voltage output sensors

If the voltage to be measured is in the range of the ADC (0–$V_{ref}$), then it can be connected directly to one of the ADC inputs. If the voltage is unipolar but can exceed $V_{ref}$, then a simple resistive voltage divider can be used to reduce the voltage to match the input range. Figure 9.1 shows the above-mentioned connections.

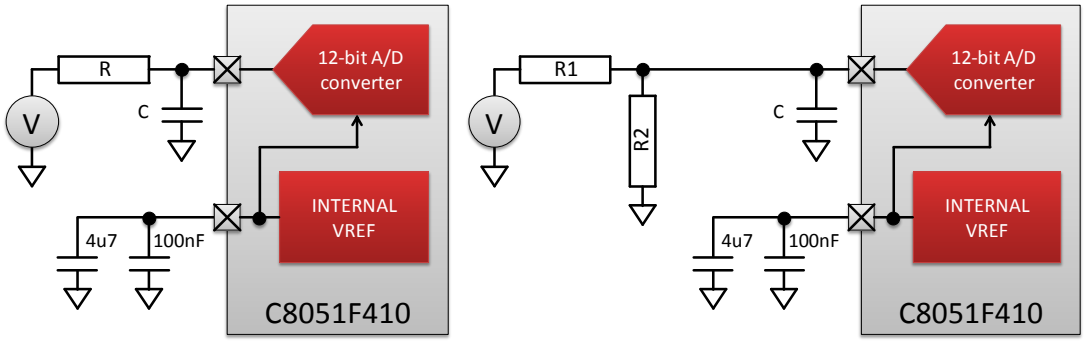


*Figure 9.1. Unipolar voltage output sensors can be connected directly or via a voltage divider to the ADC input. On the left, the ADC input voltage $V_{ADC}$ is equal to V, while on the right it is $R_2/(R_1+R_2)\cdot V$.*

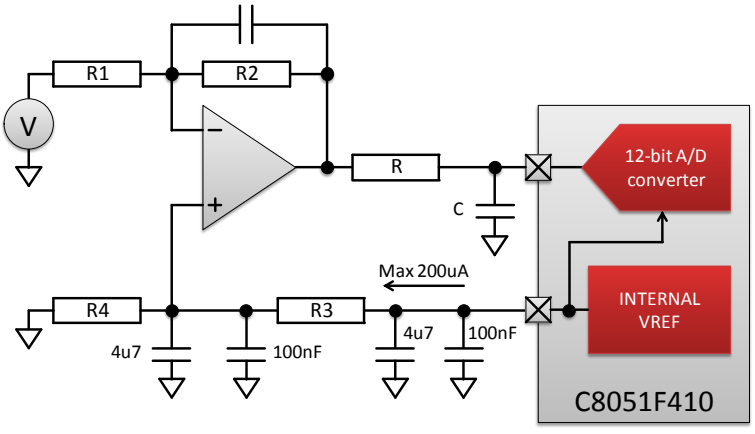


*Figure 9.2. Small or large bipolar voltages can be measured using an operational amplifier.*

If the voltage is small or bipolar, then an operational amplifier can be used to convert the signal into 0–$V_{ref}$. A general-purpose inverting circuit is shown in Figure 9.2. The output voltage of this circuit is fed to the ADC and is equal to

$$V_{\mathrm{ADC}} = \frac{V_{\mathrm{ref}} \cdot R_4}{R_3 + R_4}\left(1 + \frac{R_2}{R_1}\right) - \frac{R_2}{R_1}V \tag{9.1}$$

One can see that this formula allows small and large voltages, since the signal amplification is $-R_2/R_1$, so it can be less or greater than 1. At zero input signal, $V_{\mathrm{ADC}}$ should be close to $V_{\mathrm{ref}}/2$ for the optimal usage of the input range.

The instrumentation amplifier circuit containing three high-accuracy operational amplifiers is very useful for handling small differential sensor signals where high input impedance (no loading of the signal) is required [15]. The gain can be set by a single resistor $R_g$, and it has a level-shifting input called reference or ground. Note that the supply range of the amplifier limits the input signal range as well. Figure 9.3 shows the simplified schematic of the instrumentation amplifier.

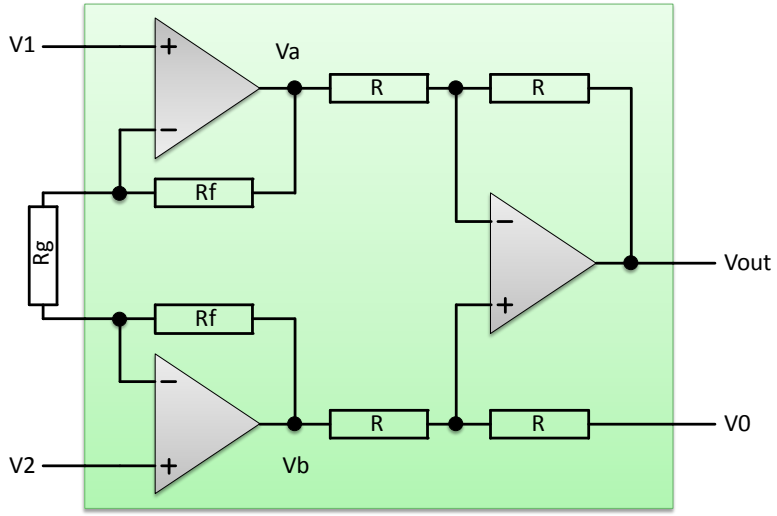


*Figure 9.3. Instrumentation amplifier (IA) circuit. Integrated IAs contain the parts drawn within the rectangle.* $V_{\mathrm{out}}=G\cdot(V_2-V_1)+V_0$, *where* $G=1+2\cdot R_f/R_g$.

The instrumentation amplifier is also available as a single integrated circuit including the low voltage AD623 amplifier, which is ideally suited to single-supply microcontroller applications. Figure 9.4 shows a typical input signal conditioning circuit using an instrumentation amplifier. Note that the operational amplifier is needed to ensure a low-impedance drive to define the middle output voltage (in our example, $V_{\mathrm{ref}}/2$) of the instrumentation amplifier.

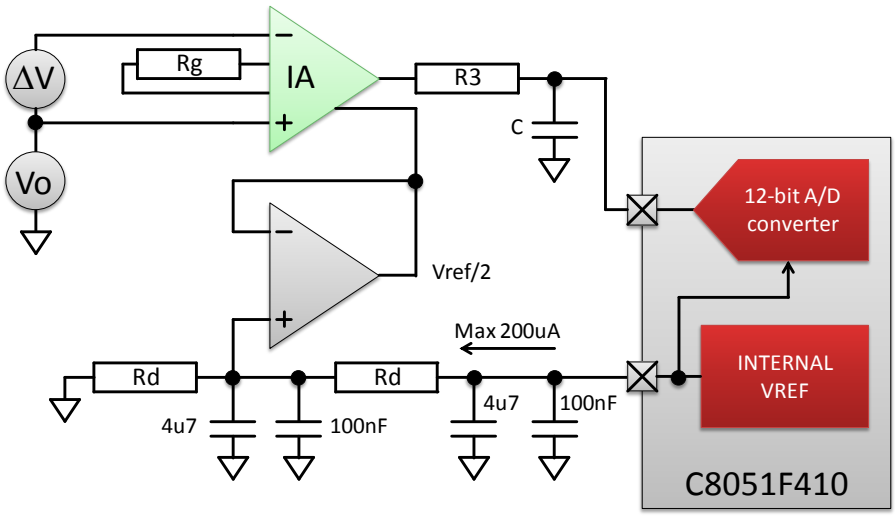


*Figure 9.4. Small voltage differences can be measured by applying an instrumentation amplifier. The ADC input voltage is $V_{ADC}=\Delta V+V_{ref}/2$.*

## 9.2 Current output sensors

Current-to-voltage conversion can be done by even a single resistor (Figure 9.5) if the current is not too high (which would cause high power dissipation) or not too low (too high impedance because of the high-value resistor). The resistor $R$ must be chosen to get a voltage equal to $V_{ref}$ when the maximum current flows.

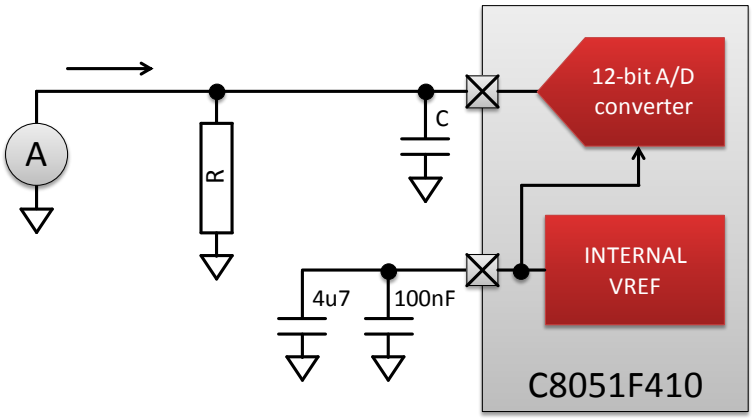


*Figure 9.5. Current-to-voltage conversion using a resistor.*

If the current is low, as in the case of a photodiode, a low input current operational amplifier based current-to-voltage converter circuit should be used; see Figure 9.6. The feedback resistor value determines the output voltage, $I \cdot R_f$.

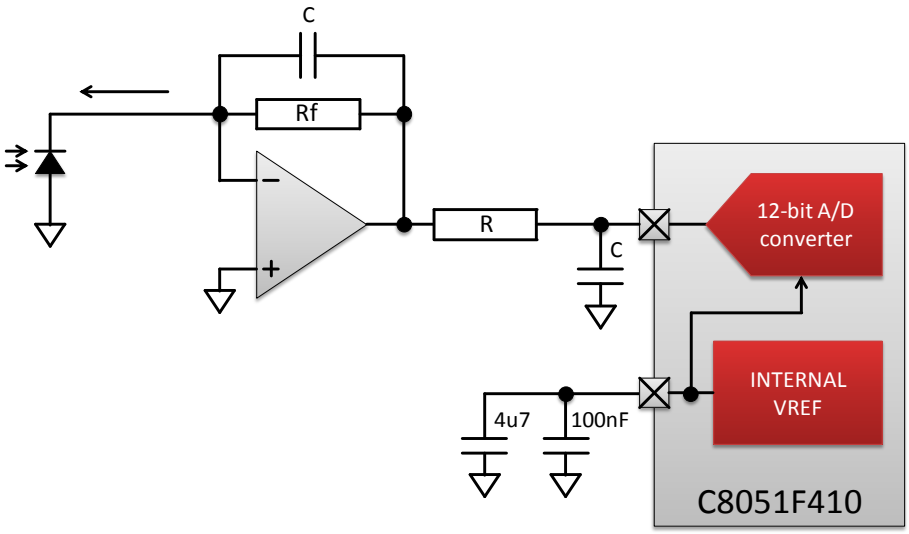


*Figure 9.6. Photodiode current-to-voltage conversion using an operational amplifier.*

Bipolar currents can be handled by simply shifting the zero-current output voltage to $V_{ref}/2$, as shown in Figure 9.7.

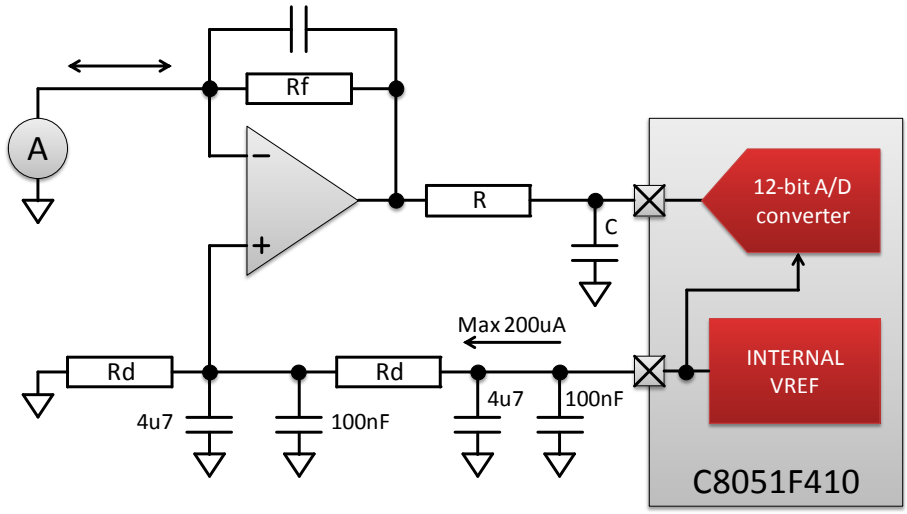


*Figure 9.7. Bipolar current-to-voltage conversion. Here the ADC input voltage $V_{ADC}$ is equal to $I \cdot R_f + V_{ref}/2$. At zero current the voltage is equal to $V_{ref}/2$.*

## 9.3 Resistive sensors

Resistive sensors, such as thermistors and photoresistors, can output a voltage if they form a resistive voltage divider with a resistor of known value (Figure 9.8). The input of the divider is the reference voltage $V_{ref}$. This circuit works in a ratiometric operation, since the ADC uses the same reference voltage as the voltage divider, so the result of the conversion does not depend on $V_{ref}$.

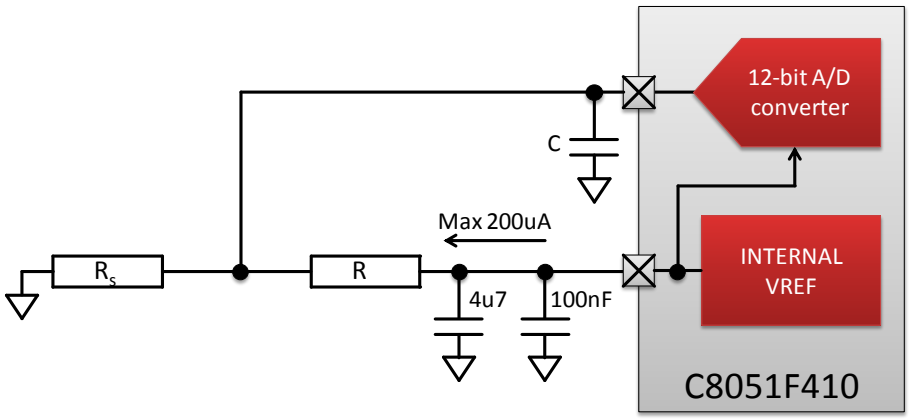


*Figure 9.8. A voltage divider allows the measurement of $R_s$. $V_{ADC}$ seen by the ADC is equal to $R_s/(R+R_s)\cdot V_{ref}$.*

Potentiometric sensors can also be connected in a very similar manner, as shown in Figure 9.9.

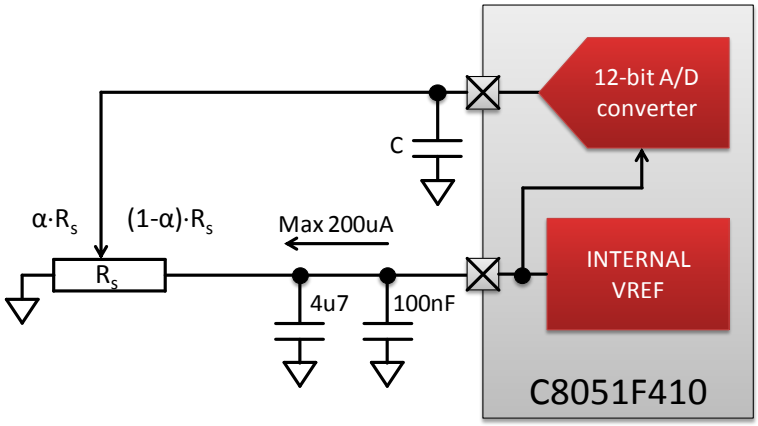


*Figure 9.9. Potentiometric sensors can be used as voltage dividers of $V_{ref}$. The ADC input voltage is $V_{ADC}=\alpha\cdot V_{ref}$.*

If the $V_{ref}$ loading were violated because of too small resistor values, the $V_{ref}$ voltage can be buffered by an operational amplifier; see Figure 9.10.

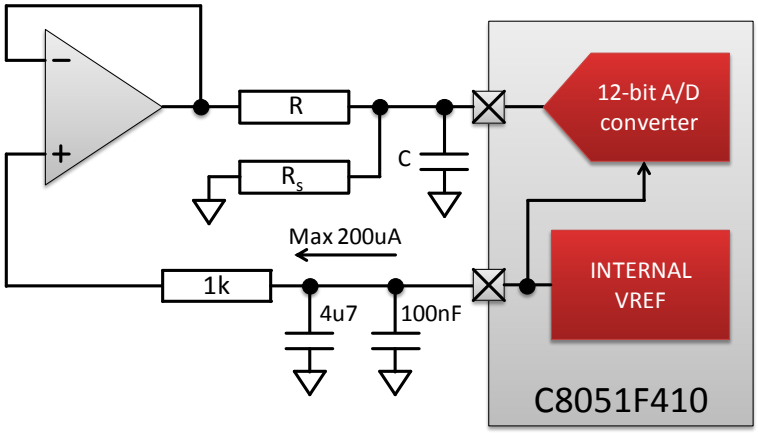


*Figure 9.10. An operational amplifier buffer removes reference loading. $V_{ADC}$ is equal to $R_s/(R+R_s)\cdot V_{ref}$.*

Pressure sensors, load cells and force sensors are typically based on a resistor bridge. The bridge can be driven by a voltage and a small differential voltage between two terminals must be measured by a high input impedance stage. The instrumentation amplifier is the ideal choice in this case, because the gain can be set by a single resistor and the output can be level-shifted by connecting a voltage – typically $V_{ref}/2$ – to its reference input. Figure 9.11 shows a possible solution.

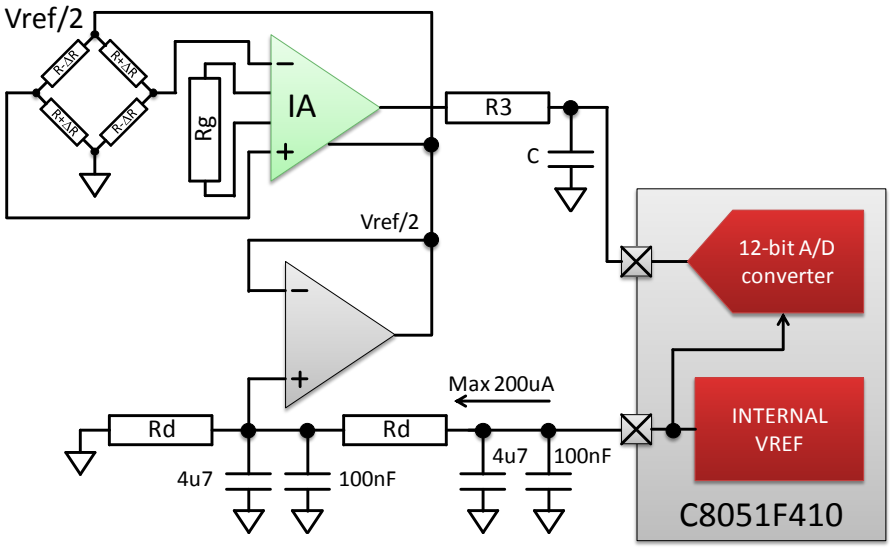


*Figure 9.11. Bridge sensors can be connected to the analogue input in ratiometric configuration using an instrumentation amplifier. The ADC input voltage is $V_{ADC}=(G\cdot\Delta R/R+1)\cdot V_{ref}/2$.*

### 9.3.1 Application guidelines

- Always consider the following in voltage measurement:
  - Voltage range. Unipolar or bipolar signal handling may be required.
  - Output impedance of the source. If it is too high, then the tracking time can be too short. Inverting amplifiers have quite a low input impedance.

### 9.3.2 Troubleshooting

**Problem:**

- Cannot communicate with the real-time clock peripheral.

**Possible reasons:**

- The interface is not opened properly. Only a reset can end the blocked state and restore normal operation.

## 9.4 Exercises

- *Design a circuit that can convert the voltage range of -10 V–10 V to 0 V–2.5 V. Check the transfer function using a circuit simulator.*
- *Design a circuit to measure the supply voltage.*

- *Design a circuit to measure the supply current.*
- *Connect a thermistor and a 10-kΩ resistor as a voltage divider of $V_{ref}$ to the ADC input. Convert the input continuously and display the temperature on the two 7-segment displays. The temperature in Celsius as a function of thermistor resistance is given by the formula $T=1/(1/298\ °C+\ln(R/R_{25})/B)-298\ °C$, where $R_{25}$ is the value of the thermistor resistance at 25 °C (nominally 10 kΩ) and B can be found in the datasheet of the thermistor (it is typically 4000 °C).*
- *Replace the thermistor with a photoresistor and display the resistance in kΩ on the two 7-segment displays.*

# 10 Real-time clock

The real-time clock circuit is a counter that is driven by an oscillator based on a tuning fork crystal with a frequency of 32768 Hz (Figure 10.1). Crystals typically have an absolute accuracy of about 10 ppm-20 ppm. For example, a 20-ppm accuracy means an error of about 1 minute in a month.

The real-time clock (called smaRTClock [6]) of the C8051F410 microcontroller uses a 47-bit binary counter that is mapped to 6 bytes (**RTC0**–**RTC5**). The least significant bit of this six-byte value is not used. This means that the four most significant bytes (**RTC2**–**RTC5**) can be considered as a 32-bit counter driven by a 1-Hz source, so it is incremented in every second. Typically the value corresponds to seconds elapsed from 0:00 January 1, 1970 and can be converted to date format using the standard C library functions declared in time.h.

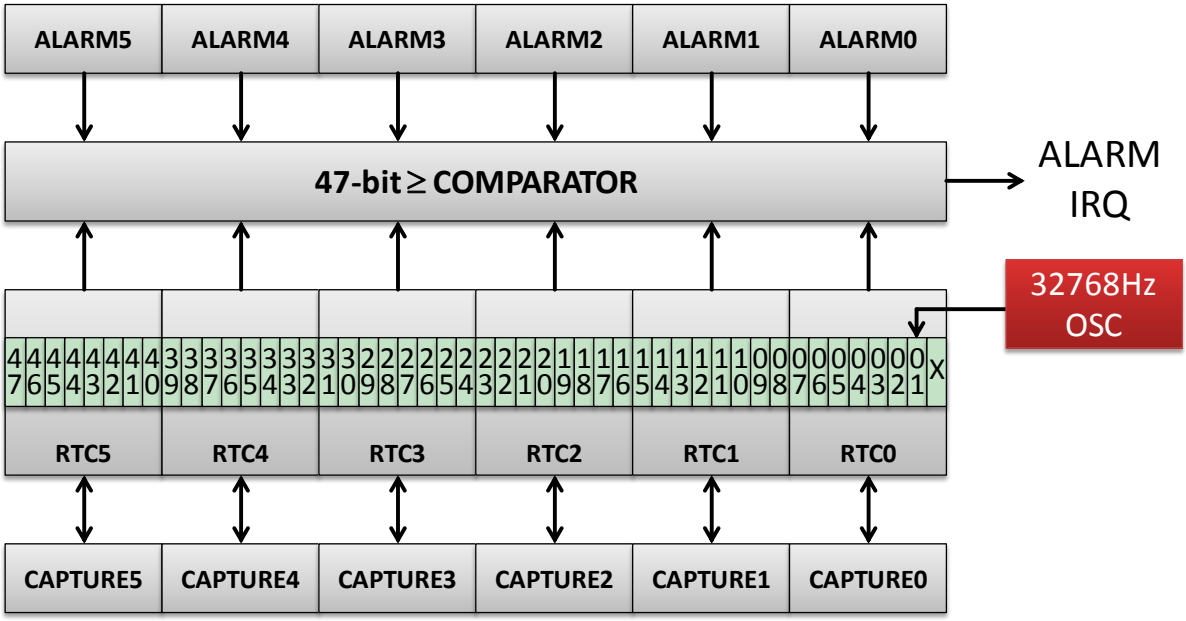


*Figure 10.1. Structure of the real-time clock.*

**CAPTURE0**–**CAPTURE5** registers are used to read and write the 47-bit counter. If it is to be written first, the **CAPTURE** registers must be loaded, then a write operation must be performed. A read transaction copies the current value to the **CAPTURE** registers for byte-wise reading. The **ALARM** registers can define a certain value that is compared to the counter and an interrupt can be generated if the values match. This can be used to wake the processor up at a certain time.

Note that if no crystal is present, the counter can also be driven by an internal oscillator, which has a selectable frequency of 20 kHz or 40 kHz. Due to its low accuracy, it cannot be used to measure real time; however, it can be useful in very low power applications when selected as the system clock.

Write or read operations of the smaRTClock registers can only be performed if the interface is opened by sending a special keyword to enable these operations. Three SFRs are available for operations: **RTC0KEY**, **RTC0ADR** and **RTC0DAT**. All internal registers can be accessed through the **RTC0ADR** address and the **RTC0DAT** data registers (see the datasheet for details).

A simple example code of basic communication functions is listed below.

```
/*************************************************************************
Unlock the smaRTClock interface
*************************************************************************/
void OpenRTC()
{
    RTC0KEY = 0xA5; // first this value must be written to the key register
    RTC0KEY = 0xF1; // next this value must be written to the key register
}

/*************************************************************************
Write to the smaRTClock registers
*************************************************************************/
void WriteRTC(unsigned char Address, unsigned char Data)
{
    while (RTC0ADR & 0x80); // wait while the smaRTClock is busy
    RTC0ADR = Address;      // set the target address
    RTC0DAT = Data;         // write the data into the register
}

/*************************************************************************
Read from the smaRTClock registers
*************************************************************************/
unsigned char ReadRTC(unsigned char Address)
{
    while (RTC0ADR & 0x80); // wait while the smaRTClock is busy
    RTC0ADR = Address;      // set the target address
    RTC0ADR|=0x80;          // define a read operation
    while (RTC0ADR & 0x80); // wait for the data
    return RTC0DAT;         // return the data
}
```

The following code initialises and starts the smaRTClock in crystal oscillator mode:

```
OpenRTC();
WriteRTC(0x07,0xE0);               // crystal mode, auto gain, double bias
WriteRTC(0x06,RTC0CN_DEF);         // power on the oscillator
for (i = 0; i < 3000; i++);        // wait a bit for stabilisation
while ((ReadRTC(0x07) & 0x10)==0); // wait for oscillator OK
WriteRTC(0x07,0xC0);               // crystal mode, auto gain, single bias
```

Note that at the beginning, the bias current of the oscillator is doubled for enhanced reliability and robustness. After stabilisation, the bias current can be reduced to normal to save power but can also be left doubled if power consumption is not a concern.

### *10.1.1 Application guidelines*

- Unlock the interface of the smaRTClock peripheral by writing 0xA5 and 0xF1, in this order, to **RTC0KEY**. If other codes are written or invalid read or write operations are initiated, the interface will be disabled until a system reset occurs.
- Select crystal oscillator mode for accurate real-time operation (an external 32768-Hz crystal must be used).
- Enable the smaRTClock crystal oscillator and wait for stabilisation by polling the corresponding bit.
- Write the initial value to the counter.
- Enable the timer by setting the timer run control bit.

- If an alarm is needed, write the corresponding counter value to the alarm registers and enable the alarm events. Provide the proper interrupt service code and enable the real time clock interrupt.
- The actual value of the counter can be set or read at any time.
- Conversion between the counter value and real time (date, hour, minute, second) can be done by the *time* and *mktime* functions defined in the standard C library (time.h).

### *10.1.2 Troubleshooting*

**Problem:**

- Cannot communicate with the real-time clock peripheral.

**Possible reasons:**

- The interface is not opened properly. Only a reset can end the blocked state and restore normal operation.

**Problem:**

- The alarm interrupt is not generated.

**Possible reasons:**

- The crystal oscillator is not running. This can be checked by reading the valid oscillation bit.
- Counting is not enabled.
- The counter value or the alarm value is invalid.
- The alarm events are not enabled or the alarm interrupt is not enabled.

## *10.2 Exercises*

- *Using the alarm function of the smaRTClock, write code that generates an interrupt in every second. Display the seconds on the 7-segment display.*
- *Write code that reads the real-time clock value and converts it to date and time using the standard C library functions.*

# 11 Watchdog and power supply monitor

In every real-world– commercial, industrial, automotive, etc. – application reliable operation is probably the most important. Microcontrollers are widely used in such embedded applications, and since they have rather complex structure, contain digital and analogue hardware components, run software when powered on, they can be sensitive to both hardware and software problems. Electromagnetic interference, spikes on the supply line, software hang-ups due to core errors, unexpected values on peripherals, infinite loops and unhandled exceptions are all potential sources that can permanently break normal operation. Since it is impossible to prevent these from occurring, some methods have been developed to safely return to normal operation.

## 11.1 The watchdog timer

One solution to avoid permanent hang-ups is the use of the so-called watchdog timer, which can detect hang-ups and reset the processor to restart normal operation. Of course, the code must be developed keeping this possibility in mind. The watchdog peripheral has an internal timer, which measures time and can reset the processor if a timeout is occurred. The timer is always restarted if the software writes to its dedicated register, so timeout will not happen if the software notifies the watchdog timer periodically within the timeout period. If any hang-up happens due to hardware or software failure, the processor will be reset within the defined timeout value.

The C8051F410 processor uses the last PCA channel to implement the watchdog timer function [6]. It is automatically enabled upon reset; therefore, the code must be developed accordingly. During prototyping and practicing the watchdog timer can be disabled, but this must be done at the beginning of the code, because otherwise a reset will be generated. Note that the C compiler generates startup code, which is executed before calling the main program and which can take longer than the default timeout period set after reset. For example, since SDCC initialises the variables by default, if a large array is declared in the external RAM space, the startup code may not be finished before a watchdog reset is generated.

In order to prevent this situation, the startup code can be redefined:

```
/**************************************************************************
Startup code redefinition
**************************************************************************/
unsigned char _sdcc_external_startup ()
{
    PCA0MD &= ~0x40; // disable watchdog timer
    PCA0MD = 0x00;   // disable watchdog timer

    VDM0CN = 0xA0;   // enable VDD monitor
    return 1;        // 1: do not initialise variables
}
```

## 11.2 Supply monitor

The supply monitor generates a reset if the supply voltage falls below the safe level. Since proper operation of digital circuits can only be guaranteed if the supply is within a certain range, unexpected behaviour may happen if the voltage gets out of this range even for a short

period. If the supply monitor is enabled, normal operation is restored by generating a reset in such cases.

### *11.2.1 Application guidelines*

- Every final version of code should use the watchdog timer and the power supply monitor to ensure reliable operation.
- During testing or code development, the watchdog timer can be disabled. It must be done at the beginning of the code to prevent undesired resets.
- The watchdog timeout can be programmed. Prefer intervals short enough to ensure quick recovery after a fault, yet long enough for the code to safely contact the watchdog timer within the timeout period in normal operation.
- The supply monitor should be switched on at the beginning of the code. After a few microseconds allowed for stabilisation, it can be enabled.

### *11.2.2 Troubleshooting*

**Problem:**

- The code does not start or unexpected resets occur.

**Possible reasons:**

- The watchdog timer is not disabled and not handled by the code.
- The watchdog timer is not restarted in time. This can be due to too short a timeout, improperly written code, time delay caused by interrupt routines or miscalculated timings.
- C compilers generate code executed before the main function, which can delay the switching off of the watchdog timer in the main function. Most compilers allow the redefining of the startup code (*_sdcc_external_startup* using the SDCC compiler), which can help to prevent this.

## *11.3 Exercises*

- *Write code that uses the watchdog timer with 1 s timeout. Try to simulate an infinite loop and check the watchdog-generated reset.*

# 12 Low-power and micropower applications

In certain cases, the microcontroller operates from a low power-supply such as a battery, solar cell or similar source. In this case, the power consumption must be kept as low as possible to meet the supply specifications and to increase battery life and reliability at the same time.

In most cases, the processor must perform operations only in a fraction of the time. Therefore, keeping the power consumption low means keeping the active operating current low and it is desirable to put the processor into an idle mode during the inactive state. Of course, some event must be used to terminate this idle mode and to resume normal operation.

## 12.1 Low-power modes

The C8051F410 processor has some low-power inactive states [6].

### 12.1.1 Idle mode

The processor can be placed in idle mode by setting the **PCON.0** high. In this mode, program execution is stopped and will be resumed if an enabled interrupt request occurs or a reset is generated. The oscillator and the peripherals are not stopped in idle mode. The supply current is reduced in idle mode: for example, the typical core supply current of 0.43 mA in normal mode at a 1-MHz system clock will be reduced to 0.21 mA in idle mode.

### 12.1.2 Stop mode

A more efficient power saving mode can be realised by the stop mode. In this mode, the internal oscillator, the core and all digital peripherals are stopped. The status of the analogue peripherals is unaffected; they can be powered down by software before entering stop mode.

An internal or external reset is required to exit from stop mode. Therefore, program execution will be restarted.

The power consumption can be very low in stop mode: the digital supply current can be as low as 0.150 μA

### 12.1.3 Suspend mode

Suspend mode is very similar to stop mode, but can be terminated by additional events including port 0 or port 1 match to a specified bit pattern, the output of an enabled comparator going low or real-time clock (smaRTClock) alarm or fail.

## 12.2 Clock speed tuning

The supply current depends on the system clock frequency in a roughly linear manner. For example, below 15 MHz the supply current can be estimated as the actual system clock frequency multiplied by 390 μA/MHz.

This allows efficient power management even without entering idle, stop or suspend modes and without stopping program execution. The system clock can be changed at any time, so it can be kept low and it is only switched to a higher frequency when more processing power is

needed. The average supply current depends on the ratio of the time spent in slower mode to that in faster mode.

The internal clock generation module of C8051F410 processor provides several different clock speeds. A 24.5-MHz internal oscillator serves as a base of the system clock generation; this value can be divided by 1, 2, 4, 8, 16, 32, 64 and 128. This means that the system clock can be as low as 191406 Hz for lowest power consumption and can be 24.5 Mhz for fastest execution. If the internal clock multiplier is used, even a system clock of 49 MHz can be generated.

Since the frequency of the internal oscillator is 24.5 Mhz regardless of the division used to generate the system clock, its power consumption is constant, typically 200 μA. In conclusion, the supply current cannot be less than this value.

Note that the real-time clock (smaRTCclock) frequency can also be selected as system clock, which allows very low supply current down to about 20 μA. The 24.5-MHz internal oscillator should be switched off to save its 200-μA operating current.

### *12.3 Peripheral power consumption*

In most low-power applications, some peripherals are used and of course they consume power. Some considerations follow concerning the power requirements of different components of the microcontroller.

**Port** pins typically drive external devices, so they may require significant current which must be considered. For example, LEDs, pull-up resistors (like those used for SMBus) and external circuitry load the ports. Note that for lowest-power operation even the internal weak pull-up resistors should be disabled.

The input clock of **digital peripherals** (such as *timers, the programmable counter array, or communication peripherals*) is derived from the system clock; therefore, their operating current is reduced if the system clock is reduced. These peripherals require significantly less power than the processor core.

**Analogue peripherals** need a certain bias current for proper operation, so they contribute to the total supply current. *Comparators* can be configured in four different power modes. Lower power can be realised at the expense of slower response. In order to reduce power consumption, the *ADC* has a special burst mode. In this mode the ADC is powered only during conversions and powered down between conversions. Therefore lowering the sample rate lowers the power required as well. Current-output *DACs* definitely provide considerable current, so if they are used, they contribute to the total supply current significantly.

### *12.4 Supply voltage*

The supply current is roughly proportional to the supply voltage of the core and of the peripherals. Since the total power dissipated by the system is equal to the supply current multiplied by the supply voltage, it is very useful to reduce the supply voltage in order to achieve low power consumption. For example, the typical supply current of the C8051F410 is 430 μA at a 2.5-V core supply voltage, which is reduced to 300 μA at 2.0 V. This means a power consumption reduction from 1.1 mW to 0.6 mW.

### *12.4.1 Application guidelines*

- The microcontroller power can be reduced using low-power modes when the core is halted. Analogue peripherals must be switched off by software. Consider the wake-up sources.
- The supply voltage should be kept low for low-power operation.
- Lower system clock frequency corresponds to lower supply current. Consider the constant current of the internal 24.5-Mhz oscillator.
- The system clock frequency can be changed during operation, but be careful: serial data transfer, timer and even ADC operation can be seriously affected.
- Minimise the loading on the port pins. Always take the current required by external components into account.
- Consider the supply current used by active digital and analogue peripherals. They should be active only during the period they are required.
- Use burst mode if the ADC is used. Keep in mind that the ADC SAR clock is derived from a dedicated 24.5-Mhz oscillator.
- Use low-power settings if comparators are used. Consider the reduced response time of the comparators.

### *12.4.2 Troubleshooting*

**Problem:**

- The supply current is significantly greater than the value given in the datasheet.

**Possible reasons:**

- The ports are loaded by external components.
- The debug adapter is connected to the system. It is safest to remove it during supply current measurement.
- Some of the active peripherals are not considered.

**Problem:**

- Invalid data are received during serial communication.

**Possible reasons:**

- The system clock frequency is changed during data transfer or the transfer speed does not match.

**Problem:**

- The ADC data seem to be invalid.

**Possible reasons:**

- The voltage reference or the ADC is powered up too close to the start of the conversion. The time is too short for accurate settling of the voltage reference, which can take several milliseconds.
- If the ADC SAR clock is too low, the internal capacitors may lose charge during conversion. Keep the ADC SAR clock as high as possible or use burst mode to avoid this problem.

## *12.5 Exercises*

- *Write code that iterates the system clock frequency upon each pressing of a button from 24.5 Mhz/128 to 24.5 MHz in a cyclic manner. Measure the digital supply current as a function of the clock frequency. Consider any possible loads on the port pins (including the debug adapter).*
- *Write code that wakes up the microcontroller in every second from a suspend state using the smaRTClock alarm function. The code must switch an LED on for 100 ms then should go back to suspend mode.*
- *Write code that wakes the microcontroller up from a suspend state if a button has been pressed. The code must switch an LED on for 100 ms then should go back to suspend mode. Use the port match event to detect button pressings and to terminate the suspend state.*

# 13 USB, wired and wireless communications

Most microcontrollers do not have communication interfaces that support direct connection to personal computers or host computers. The most popular wired interface is the USB port, which can even provide power supply for the connected peripheral. Devices can be wirelessly connected via a Bluetooth module especially developed for low-power small device applications.

There are microcontrollers with built-in USB interfaces or wireless communication modules, but they only represent a fraction of the wide selection of microcontrollers with a rich set of analogue and digital peripherals.

A more general solution is to use a USB-UART, Bluetooth or other wireless module connected to the UART or similar port available on all microcontrollers. Somewhat more space and at least two integrated circuits are required, but in this case, practically any microcontroller can be used, which guarantees exceptional flexibility.

## 13.1 USB-UART interfaces

One of the most popular and most reliable USB-UART converters is the FT232R [19]. The chip can be connected to the UART port and can handle the quite complicated USB protocol. Only a few external capacitors are needed as power supply decoupling capacitors. The FT232R chip supports full-speed USB communication (12 Mbit/s); however, baud rates are limited to a maximum of 3 Mbit/s. Sending a byte means sending a start bit, 8 data bits and 1-2 stop bits, so the achievable throughput is somewhat below 300 kbyte/s. The FT232R contains a 25-byte FIFO (first in-first out) buffer memory to avoid data loss at high data rates.

Note that since downstream data must be directly received by the microcontroller from the FT232R chip, the transmit FIFO of the FT232R cannot be used. Therefore, a software FIFO must be implemented in the microcontroller code at high speed transfers. See the UART interrupt mode examples in Chapter 7.2.

The host computer can communicate with the microcontroller via the native driver or via the virtual COM port driver, which is easy to use even with a simple terminal software and easy to program in C, C++, C#, Java, LabVIEW or Matlab.

Note that the virtual COM port mode has limited configuration possibilities. For example, the so-called latency time cannot be set and its default value is 16 ms. This means that if the host wants to send only a few bytes (at least less than the buffer size to trigger an USB transmit transaction), then the latency time must elapse before sending the data. This can slow communication down, so it is recommended to set the latency time to its minimum, 1 ms, using the hardware configuration utility of the operating system.

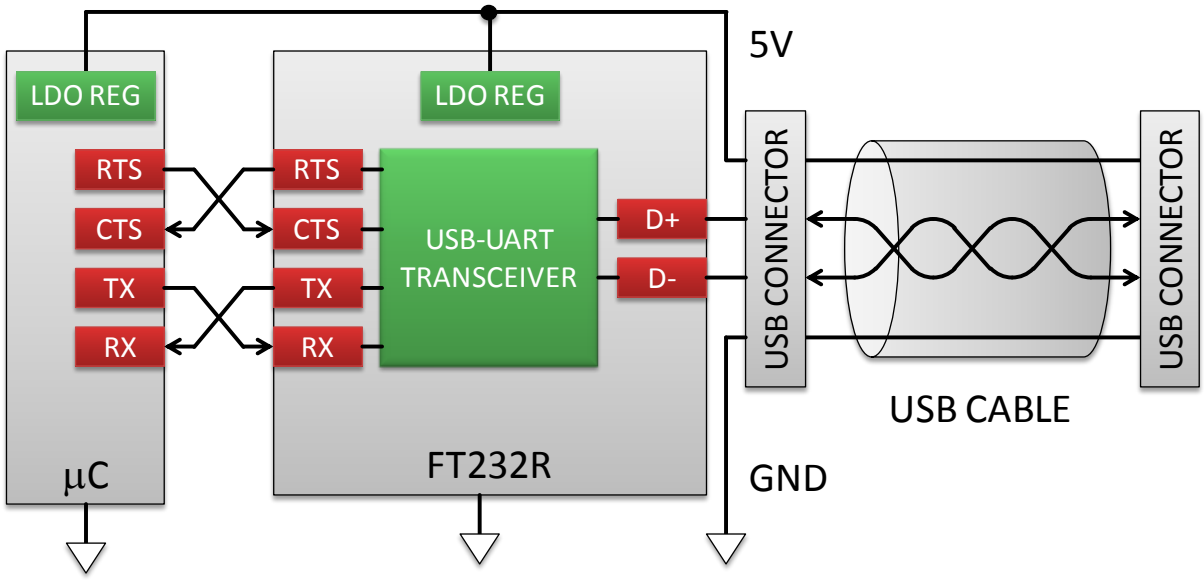


*Figure 13.1. Connecting a microcontroller to a USB port using the FT232R USB-UART converter.*

Figure 13.1 shows how the microcontroller can be connected to a USB port using the FT232R USB-UART converter. The TX and RX are the UART port bits, while the RTS (ready-to-send) and CTS (clear-to-send) on the microcontroller are provided by general-purpose port bits. These lines are optional and can be used for handshaking – checking if data is available or if the receiver is ready to accept data. Note that the USB port can even power the circuit; the low dropout regulators (LDO REG) output the required supply voltage that is normally less than 5 V.

The complete example schematic and board layout can be seen in Figure 13.2 and Figure 13.3.

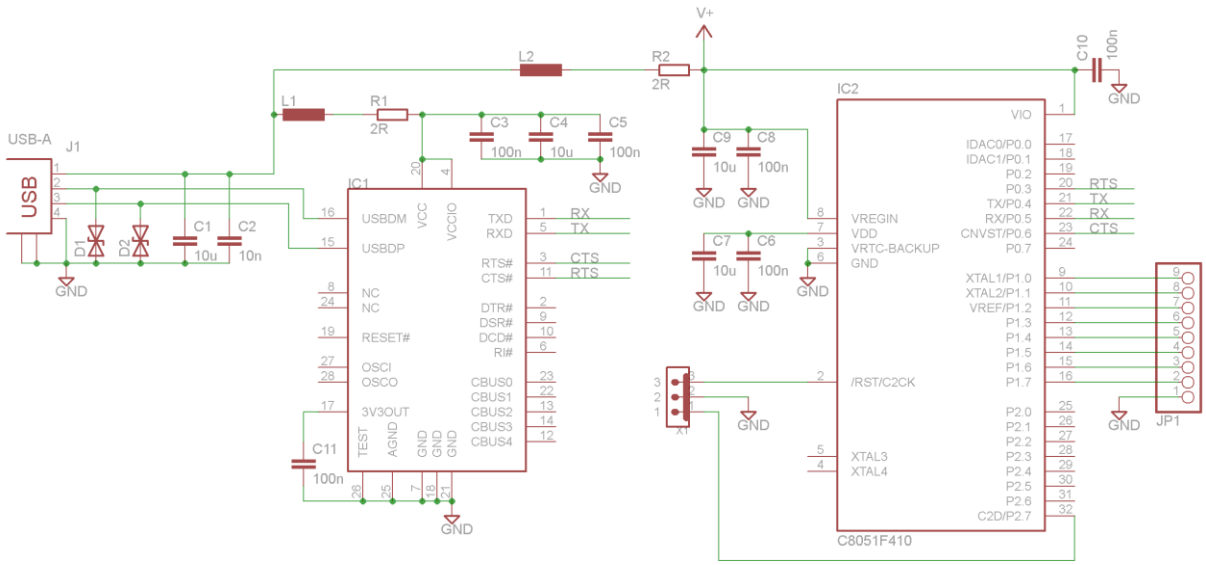


*Figure 13.2. Schematic of the C8051F410 microcontroller USB interface.*

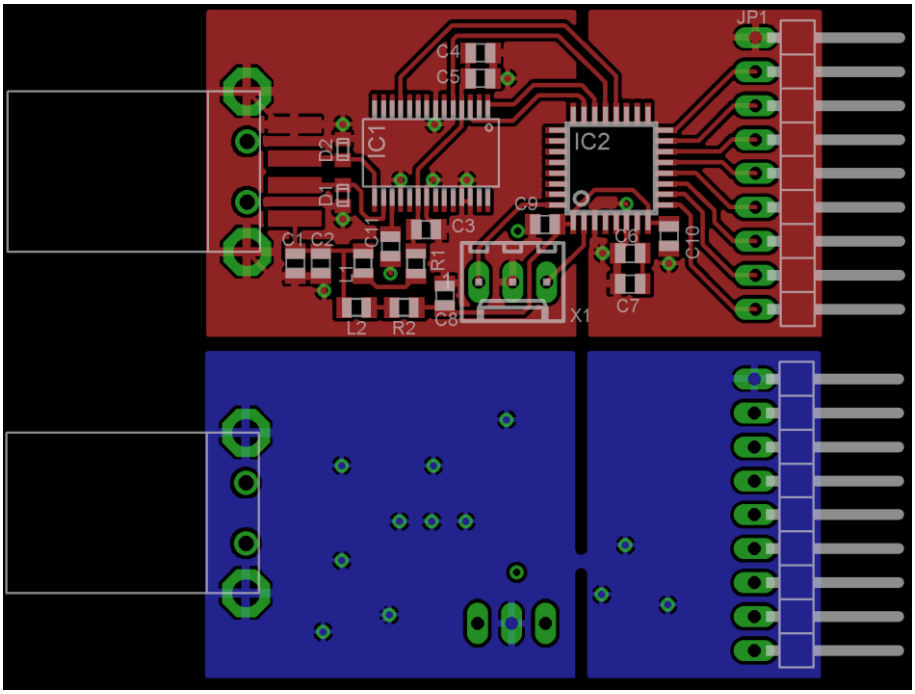


*Figure 13.3. Component (red) and bottom (blue) side of the C8051F410 microcontroller USB interface printed circuit board.*

All supply lines are decoupled with ceramic chip capacitors placed as close to the supply pins as possible. The bottom side realises the required solid ground plane, and the signal ground and USB grounds are connected at the microcontroller. This separates the sensitive analogue circuitry of the microcontroller from the noisy ground return currents of the digital part. D1 and D2 are USB data line protection diodes, X1 is the debug port and JP1 is a connector for port **P1**. This port can accept both analogue and digital input or output signals depending on the configuration of the port **P1**.

Note that there are faster (USB 2.0) USB-UART interfaces, including the FT2232H, which uses the same drivers on the computer side and therefore can be used to seamlessly upgrade communication speed. However, the microcontroller bit rate is limited, so only high clock frequency microcontrollers can benefit from this solution.

USB-to-parallel interfaces can also be used to transfer a whole byte at a time. This provides the fastest communication at the expense of more complex circuitry and of the fact that much more pins of the microcontroller must be used.

## *13.2 Wireless communication possibilities*

There are small wireless modules that can be also connected to the microcontroller. Bluetooth modules are widely available and have a standard SPP (serial port protocol) mode to be driven directly from a UART port of a device. After setting up the module, it will be fully transparent: a virtual COM port on the host computer can be used in the same way as for the FT232R USB-UART converter or a regular COM port. In such cases, even smart phones can be used to easily communicate with the microcontroller-based hardware unit.

## *13.3 Exercises*

- *Write code that measures the state of the potentiometer and sends the data in text format over the UART using a 9600-bit/s baud rate. Check the result with terminal software using the virtual COM port.*
- *Write code that measures the state of the potentiometer and sends the data in text format over the BTM-112 Bluetooth module. Check the result with terminal software using the virtual COM port.*
- *Write code that measures the state of the in-chip temperature and sends the data in text format over the BTM-112 Bluetooth module. Check the result with terminal software running on a smart phone.*

# 14 Development kit

## 14.1 The C8051F410 development kit

The C8051F410 development kit is manufactured by Silicon Laboratories to support rapid development and testing [7]. It can be used as a general-purpose platform to develop many different microcontroller applications. The board is powered from a wall-plug adapter and integrates LEDs, push buttons, a serial host interface, a potentiometer, a watch crystal, a battery socket and even more. The complete description can be found in the user manual that can be downloaded from the manufacturer's pages.

The board has a two-row pin header connector that allows access of any port pin of the microcontroller and supports the connection of various external circuitries. An extension board with 6 additional LEDs, two 7-segment displays, a 3-pin general purpose analogue sensor port and an LM75 temperature sensor is shown in the photo in Figure 15.1.

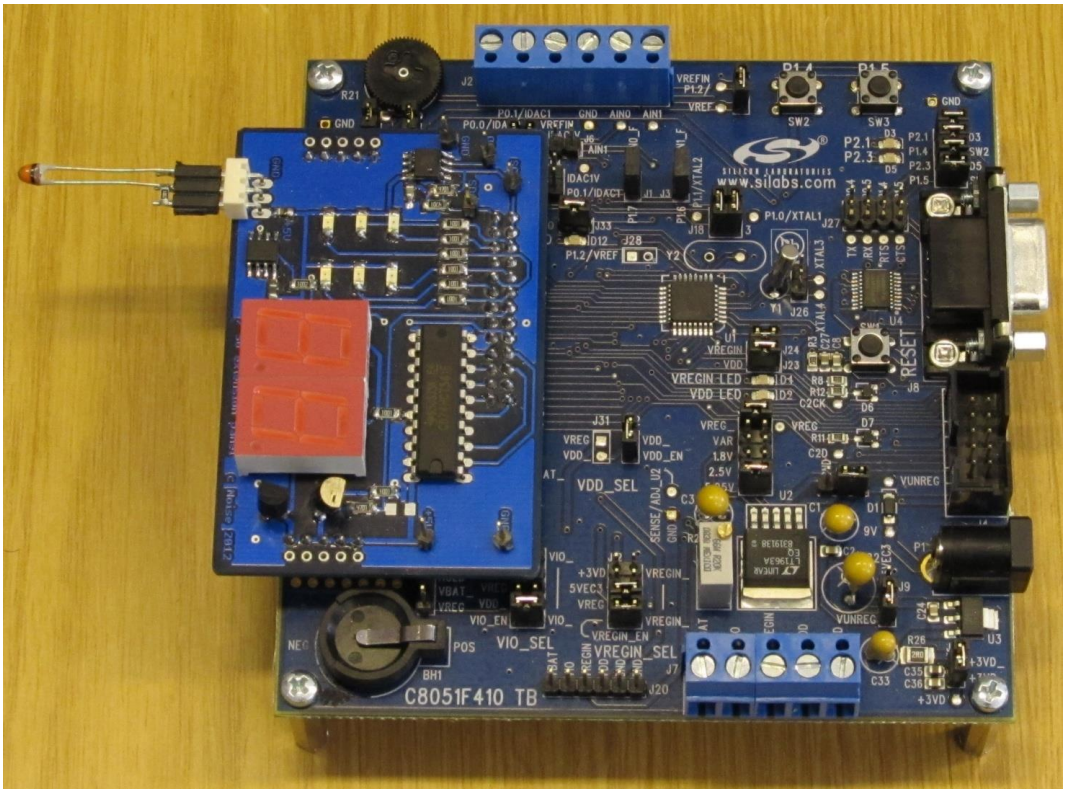


*Figure 15.1. The C8051F410TB target board with the extension board. On the left side, a thermistor connected to the general-purpose analogue input can be seen*

The extension board is documented in the next chapter.

## 14.2 Extension board

The extension board is a powerful supplement to the C8051F410 development kit. It can be used to practice many features of the microcontroller, while it also serves as a reference design.

The six LEDs are driven from pins of port **P0** and **P1**. The anodes of the LEDs are connected to the positive supply, so both open-drain or push-pull mode can be used to light them. The

current limiting resistors have a value of 1 kΩ, which ensures proper light intensity. The LEDS are arranged on the board as two traffic lights and their colours are red, yellow and green. This supports practicing several related applications. Note that the LEDs form a six-point rectangle (or circle), so, for example, stepper motor control can also be simulated and visualised.

Two 7-segment displays are connected to port **P2** via a buffer to reduce the total port current of the microcontroller. Port bit **P1.3** is used to select which 7-segment display is active. Both displays cannot be used at the same time; however, this can be used to demonstrate how a fast alternation of the displays can be applied to implement a simultaneous-looking display of two digits. The display therefore can be used to count from 0 to 99, implement a second counter or display a temperature in degrees, etc.

The U$3 and U$4 pin headers are only used to connect the ground and the positive supply to the extension boards.

An LM75 I²C temperature sensor is connected to port pins **P0.0** and **P0.1**. This supports the measurement of external temperature and also allows the learning of the use of the SMBus/I²C interface.

The 3-pin header labelled IN1 is a general-purpose analogue and sensor interface. The three pins are connected to the system ground, the 5 V supply and a high impedance input of a rail-to-rail input and output operational amplifier. The output of this operational amplifier is connected to pin **P1.7** of the microcontroller via a voltage divider and the voltage can be measured by the internal A/D converter. This allows the measurement of voltage-output sensors (for example, Hall effect magnetic field sensors), resistive sensors (such as light-dependent resistors, thermistors, etc.). Current-output sensors can also be connected if an external current-to-voltage conversion resistor is connected in parallel with the sensor. The 5 V supply can serve as a supply for active sensors or can be used as the input voltage of a voltage divider formed by a resistor of known value and a resistive sensor. See Chapter 9 for more information about connecting sensors to the microcontroller.

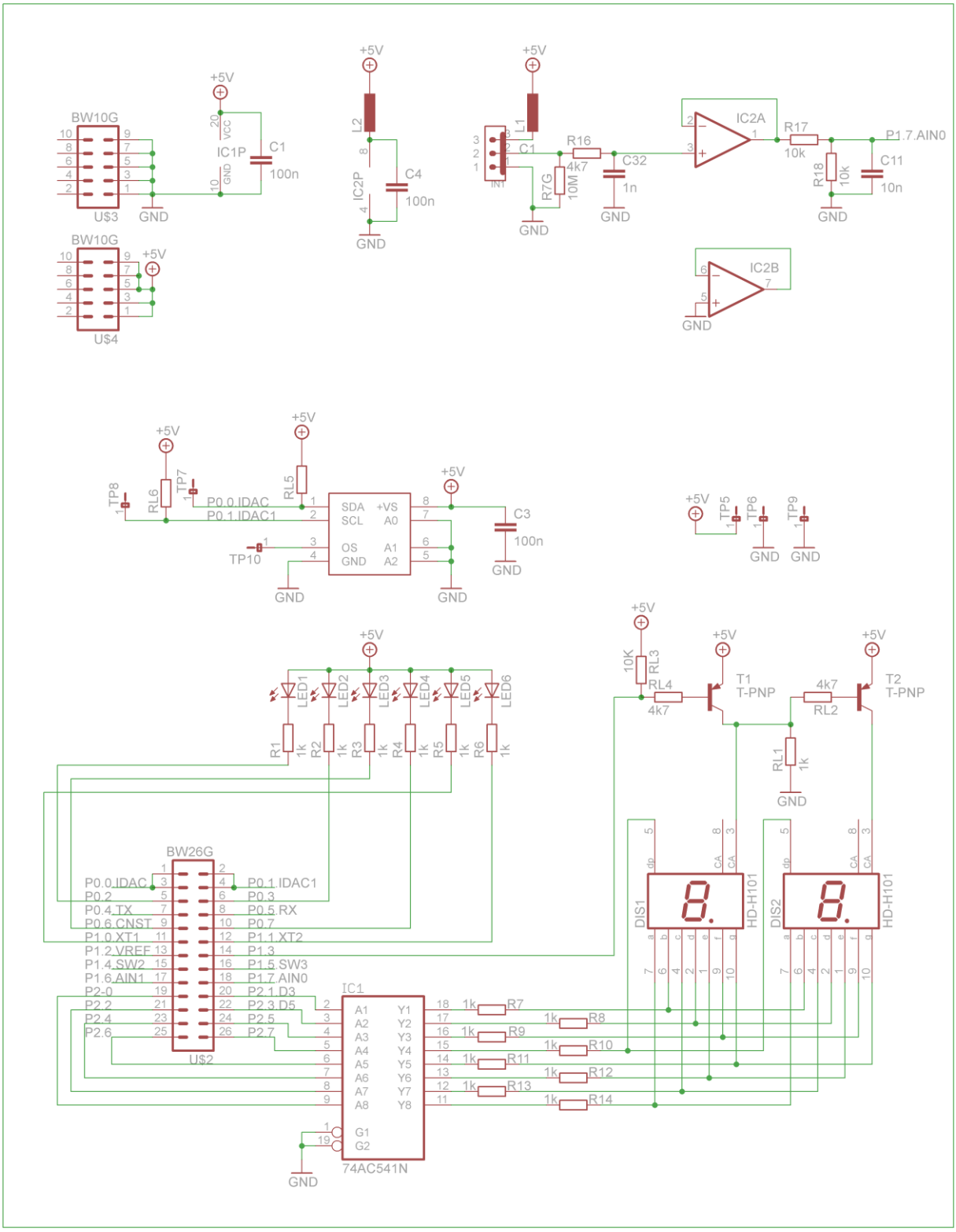


*Figure 15.2. The extension board schematic.*

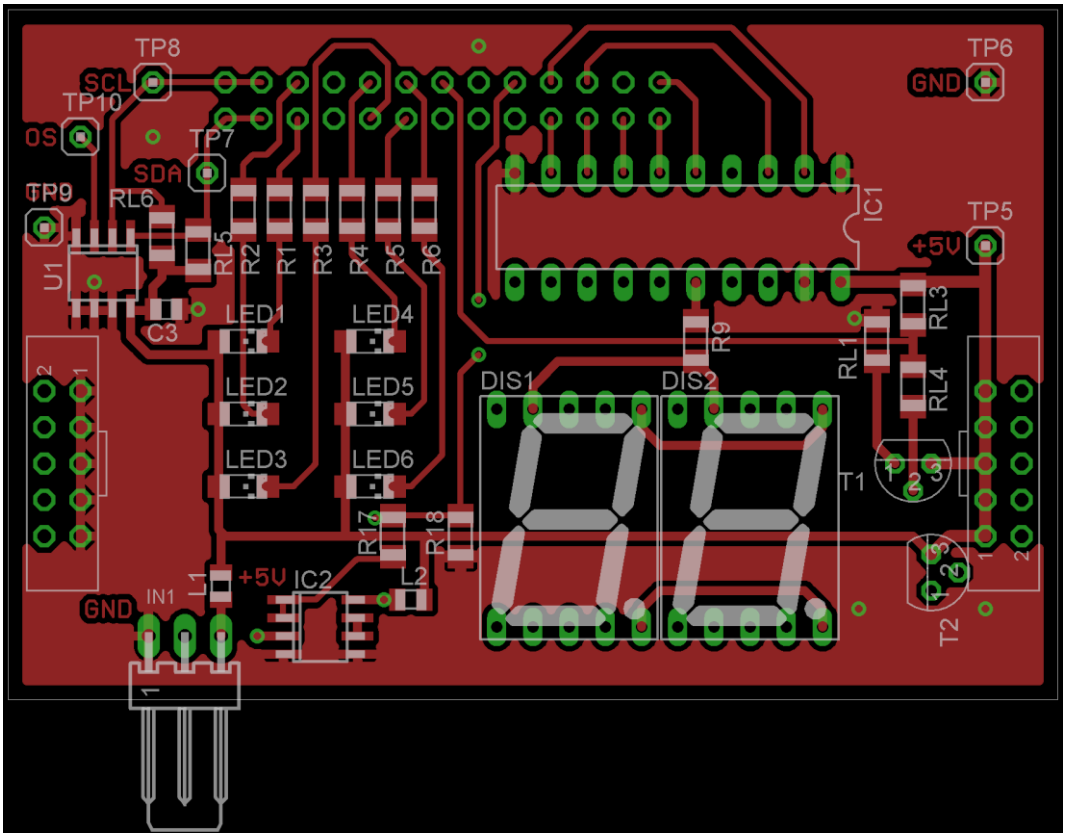


*Figure 15.3. Extension board top side layout.*

# 15 Acknowledgements

The work has been supported by the European Union and co-funded by the European Social Fund, project number: TÁMOP-4.1.2.A/1-11/1.

We thank Silicon Laboratories and their local distributor HT-Eurep Ltd. (Hungary) for providing the development kits to support education. The technical documents, application notes, knowledge base and user forum of Silicon Laboratories provided very valuable help in our work.

We are grateful to the reviewers Dr. György Györök and Dr. Péter Makra who read the manuscript carefully; they corrected several errors and recommended changes also.

# 16 References


[1] Chew Moi Tin, Gourab Sen Gupta: Embedded Programming with Field-Programmable Mixed-Signal µControllers,
Silicon Laboratories MCUniversity Course Material
http://www.silabs.com/products/mcu/Pages/MCUniversity.aspx

[2] 80c51 family architecture, NXP (Philips Semiconductor)
http://www.lpcware.com/content/nxpfile/80c51-family-architecture
80c51 family programmers guide and instruction set, NXP (Philips Semiconductor)
http://www.lpcware.com/content/nxpfile/80c51-family-programmers-guide-and-instruction-set
80c51 family hardware description, NXP (Philips Semiconductor)
http://www.lpcware.com/content/nxpfile/80c51-family-hardware-description

[3] Keil 51 Assembler,
http://www.keil.com/c51/a51kit.asp

[4] SDCC (Small Device C Compiler) User Manual,
http://sdcc.sourceforge.net/

[5] I. Scott MacKenzie, The 8051 Microcontroller (3rd Edition), Prentice Hall, 1998

[6] C8051F410 datasheet, Silicon Laboratories, 2008

[7] C8051F41x-DK User Guide, Silicon Laboratories, 2006

[8] Silicon Laboratories Application Notes,
http://www.silabs.com/products/mcu/Pages/ApplicationNotes.aspx

[9] Silicon Laboratories Knowledge Base,
http://www.silabs.com/support/knowledgebase/pages/default.aspx

[10] Silicon Laboratories User Forum,
http://www.silabs.com/support/forums/Pages/default.aspx

[11] W. Jung, Op Amp Applications Handbook, Newnes. 2006.,
http://www.analog.com/library/analogDialogue/archives/39-05/op_amp_applications_handbook.html

[12] W. Kester, The Data Conversion Handbook, Newnes, 2005.,
http://www.analog.com/library/analogDialogue/archives/39-06/data_conversion_handbook.html

[13] W. Kester, Mixed-Signal and DSP Design Techniques, Newnes, 2003.,
http://www.analog.com/en/content/mixed_signal_dsp_design_book/fca.html

[14] H. Zumbahlen , Linear Circuit Design Handbook, Newnes, 2008.,
http://www.analog.com/library/analogDialogue/archives/43-09/linear_circuit_design_handbook.html

[15] C. Kitchin and L. Counts, A Designer's Guide to Instrumentation Amplifiers 3rd edition, Analog Devices, Inc. 2006.,
http://www.analog.com/en/power-management/power-monitors/ad8557/products/CU_dh_designers_guide_to_instrumentation_amps/fca.html

[16] HD44780 datasheet, see for example http://lcd-linux.sourceforge.net/pdfdocs/hd44780.pdf

[17] 80C51FA/FB PCA Cookbook, Intel application note AP-415.

[18] Determining Clock Accuracy Requirements for UART Communications,
http://pdfserv.maximintegrated.com/en/an/AN2141.pdf

[19] http://www.ftdichip.com/