A **microcontroller** (sometimes abbreviated μC, uC or MCU) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications.

The die from an Intel 8742, an 8-bit microcontroller that includes a CPU running at 12 MHz, 128 bytes of RAM, 2048 bytes of EPROM, and I/O in the same chip.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems.

Some microcontrollers may use four-bit words and operate at clock rate frequencies as low as 4 kHz, for low power consumption (milliwatts or microwatts). They will generally have the ability to retain functionality while waiting for an event such as a button press or other interrupt; power consumption while sleeping (CPU clock and most peripherals off) may be just nanowatts, making many of them well suited for long lasting battery applications. Other microcontrollers may serve performance-critical roles, where they may need to act more like a digital signal processor (DSP), with higher clock speeds and power consumption.
Embedded design

A microcontroller can be considered a self-contained system with a processor, memory and peripherals and can be used as an embedded system.[1] The majority of microcontrollers in use today are embedded in other machinery, such as automobiles, telephones, appliances, and peripherals for computer systems. These are called embedded systems. While some embedded systems are very sophisticated, many have minimal requirements for memory and program length, with no operating system, and low software complexity. Typical input and output devices include switches, relays, solenoids, LEDs, small or custom LCD displays, radio frequency devices, and sensors for data such as temperature, humidity, light level etc. Embedded systems usually have no keyboard, screen, disks, printers, or other recognizable I/O devices of a personal computer, and may lack human interaction devices of any kind.

Interrupts

Microcontrollers must provide real time (predictable, though not necessarily fast) response to events in the embedded system they are controlling. When certain events occur, an interrupt system can signal the processor to suspend processing the current instruction sequence and to begin an interrupt service routine (ISR, or "interrupt handler"). The ISR will perform any processing required based on the source of the interrupt before returning to the original instruction sequence. Possible interrupt sources are device dependent, and often include events such as an internal timer overflow, completing an analog to digital conversion, a logic level change on an input such as from a button being pressed, and data received on a communication link. Where power consumption is important as in battery operated devices, interrupts may also wake a microcontroller from a low power sleep state where the processor is halted until required to do something by a peripheral event.

Programs

Microcontroller programs must fit in the available on-chip program memory, since it would be costly to provide a system with external, expandable, memory. Compilers and assemblers are used to convert high-level language and assembler language codes into a compact machine code for storage in the microcontroller's memory. Depending on the device, the program memory may be permanent, read-only memory that can only be programmed at the factory, or program memory may be field-alterable flash or erasable read-only memory.

Other microcontroller features

Microcontrollers usually contain from several to dozens of general purpose input/output pins (GPIO). GPIO pins are software configurable to either an input or an output state. When GPIO pins are configured to an input state, they are often used to read sensors or external signals. Configured to the output state, GPIO pins can drive external devices such as LEDs or motors.
Many embedded systems need to read sensors that produce analog signals. This is the purpose of the analog-to-digital converter (ADC). Since processors are built to interpret and process digital data, i.e. 1s and 0s, they are not able to do anything with the analog signals that may be sent to it by a device. So the analog to digital converter is used to convert the incoming data into a form that the processor can recognize. A less common feature on some microcontrollers is a digital-to-analog converter (DAC) that allows the processor to output analog signals or voltage levels.

In addition to the converters, many embedded microprocessors include a variety of timers as well. One of the most common types of timers is the Programmable Interval Timer (PIT). A PIT may either count down from some value to zero, or up to the capacity of the count register, overflowing to zero. Once it reaches zero, it sends an interrupt to the processor indicating that it has finished counting. This is useful for devices such as thermostats, which periodically test the temperature around them to see if they need to turn the air conditioner on, the heater on, etc.

Time Processing Unit (TPU) is a sophisticated timer. In addition to counting down, the TPU can detect input events, generate output events, and perform other useful operations.

A dedicated Pulse Width Modulation (PWM) block makes it possible for the CPU to control power converters, resistive loads, motors, etc., without using lots of CPU resources in tight timer loops.

Universal Asynchronous Receiver/Transmitter (UART) block makes it possible to receive and transmit data over a serial line with very little load on the CPU. Dedicated on-chip hardware also often includes capabilities to communicate with other devices (chips) in digital formats such as I2C and Serial Peripheral Interface (SPI).

**Higher integration**

Micro-controllers may not implement an external address or data bus as they integrate RAM and non-volatile memory on the same chip as the CPU. Using fewer pins, the chip can be placed in a much smaller, cheaper package.

Integrating the memory and other peripherals on a single chip and testing them as a unit increases the cost of that chip, but often results in decreased net cost of the embedded system as a whole. Even if the cost of a CPU that has integrated peripherals is slightly more than the cost of a CPU and external peripherals, having fewer chips typically allows a smaller and cheaper circuit board, and reduces the labor required to assemble and test the circuit board.

A micro-controller is a single integrated circuit, commonly with the following features:

- central processing unit - ranging from small and simple 4-bit processors to complex 32- or 64-bit processors
- volatile memory (RAM) for data storage
- ROM, EPROM, EEPROM or Flash memory for program and operating parameter storage
- discrete input and output bits, allowing control or detection of the logic state of an individual package pin
- serial input/output such as serial ports (UARTs)
- other serial communications interfaces like I²C, Serial Peripheral Interface and Controller Area Network for system interconnect
- peripherals such as timers, event counters, PWM generators, and watchdog
- clock generator - often an oscillator for a quartz timing crystal, resonator or RC circuit
- many include analog-to-digital converters, some include digital-to-analog converters
- in-circuit programming and debugging support

This integration drastically reduces the number of chips and the amount of wiring and circuit board space that would be needed to produce equivalent systems using separate chips. Furthermore, on low pin count devices in particular, each pin may interface to several internal peripherals, with the pin function selected by software. This allows a part to be used in a wider variety of applications than if pins had dedicated functions. Microcontrollers have proved to be highly popular in embedded systems since their introduction in the 1970s.

Some microcontrollers use a Harvard architecture: separate memory buses for instructions and data, allowing accesses to take place concurrently. Where a Harvard architecture is used, instruction words for the processor may be a different bit size than the length of internal memory and registers; for example: 12-bit instructions used with 8-bit data registers.

The decision of which peripheral to integrate is often difficult. The microcontroller vendors often trade operating frequencies and system design flexibility against time-to-market requirements from their customers and overall lower system cost. Manufacturers have to balance the need to minimize the chip size against additional functionality.

Microcontroller architectures vary widely. Some designs include general-purpose microprocessor cores, with one or more ROM, RAM, or I/O functions integrated onto the package. Other designs are purpose built for control applications. A micro-controller instruction set usually has many instructions intended for bit-wise operations to make control programs more compact.\[2\] For example, a general purpose processor might require several instructions to test a bit in a register and branch if the bit is set, where a micro-controller could have a single instruction to provide that commonly-required function.

Microcontrollers typically do not have a math coprocessor, so floating point arithmetic is performed by software.
Volumes

About 55% of all CPUs sold in the world are 8-bit microcontrollers and microprocessors. According to Semico, over four billion 8-bit microcontrollers were sold in 2006.[3]

A typical home in a developed country is likely to have only four general-purpose microprocessors but around three dozen microcontrollers. A typical mid-range automobile has as many as 30 or more microcontrollers. They can also be found in many electrical devices such as washing machines, microwave ovens, and telephones.

![A PIC 18F8720 microcontroller in an 80-pin TQFP package.](image)

Manufacturers have often produced special versions of their microcontrollers in order to help the hardware and software development of the target system. Originally these included EPROM versions that have a "window" on the top of the device through which program memory can be erased by ultraviolet light, ready for reprogramming after a programming ("burn") and test cycle. Since 1998, EPROM versions are rare and have been replaced by EEPROM and flash, which are easier to use (can be erased electronically) and cheaper to manufacture.

Other versions may be available where the ROM is accessed as an external device rather than as internal memory, however these are becoming increasingly rare due to the widespread availability of cheap microcontroller programmers.

The use of field-programmable devices on a microcontroller may allow field update of the firmware or permit late factory revisions to products that have been assembled but not yet shipped. Programmable memory also reduces the lead time required for deployment of a new product.

Where hundreds of thousands of identical devices are required, using parts programmed at the time of manufacture can be an economical option. These "mask programmed" parts have the program laid down in the same way as the logic of the chip, at the same time.
Programming environments

Microcontrollers were originally programmed only in assembly language, but various high-level programming languages are now also in common use to target microcontrollers. These languages are either designed specially for the purpose, or versions of general purpose languages such as the C programming language. Compilers for general purpose languages will typically have some restrictions as well as enhancements to better support the unique characteristics of microcontrollers. Some microcontrollers have environments to aid developing certain types of applications. Microcontroller vendors often make tools freely available to make it easier to adopt their hardware.

Many microcontrollers are so quirky that they effectively require their own non-standard dialects of C, such as SDCC for the 8051, which prevent using standard tools (such as code libraries or static analysis tools) even for code unrelated to hardware features. Interpreters are often used to hide such low level quirks.

Interpreter firmware is also available for some microcontrollers. For example, BASIC on the early microcontrollers Intel 8052; BASIC and FORTH on the Zilog Z8 as well as some modern devices. Typically these interpreters support interactive programming.

Simulators are available for some microcontrollers, such as in Microchip's MPLAB environment and the Revolution Education PICAXE range. These allow a developer to analyze what the behavior of the microcontroller and their program should be if they were using the actual part. A simulator will show the internal processor state and also that of the outputs, as well as allowing input signals to be generated. While on the one hand most simulators will be limited from being unable to simulate much other hardware in a system, they can exercise conditions that may otherwise be hard to reproduce at will in the physical implementation, and can be the quickest way to debug and analyze problems.

Recent microcontrollers are often integrated with on-chip debug circuitry that when accessed by an in-circuit emulator via JTAG, allow debugging of the firmware with a debugger.

Types of microcontrollers

List of common microcontrollers

As of 2008 there are several dozen microcontroller architectures and vendors including:

- Parallax Propeller
- Freescale 68HC11 (8-bit)
- Intel 8051
- Silicon Laboratories Pipelined 8051 Microcontrollers
- ARM processors (from many vendors) using ARM7 or Cortex-M3 cores are generally microcontrollers
• STMicroelectronics (8-bit), ST10 (16-bit) and STM32 (32-bit)
• Atmel AVR (8-bit), AVR32 (32-bit), and AT91SAM (32-bit)
• Freescale ColdFire (32-bit) and S08 (8-bit)
• Hitachi H8, Hitachi SuperH (32-bit)
• Infineon Microcontroller: 8, 16, 32 Bit microcontrollers for automotive and industrial applications
• MIPS (32-bit PIC32)
• NEC V850 (32-bit)
• NXP Semiconductors LPC1000, LPC2000, LPC3000, LPC4000 (32-bit), LPC900, LPC700 (8-bit)
• PIC (8-bit PIC16, PIC18, 16-bit dsPIC33 / PIC24)
• PowerPC ISE
• PSoC (Programmable System-on-Chip)
• Rabbit 2000 (8-bit)
• Texas Instruments Microcontrollers : TI MSP430 16-bit Microcontrollers
• Toshiba TLCS-870 (8-bit/16-bit)

and many others, some of which are used in very narrow range of applications or are more like applications processors than microcontrollers. The microcontroller market is extremely fragmented, with numerous vendors, technologies, and markets. Note that many vendors sell (or have sold) multiple architectures.

**Interrupt latency**

In contrast to general-purpose computers, microcontrollers used in embedded systems often seek to optimize interrupt latency over instruction throughput. Issues include both reducing the latency, and making it be more predictable (to support real-time control).

When an electronic device causes an interrupt, the intermediate results (registers) have to be saved before the software responsible for handling the interrupt can run. They must also be restored after that software is finished. If there are more registers, this saving and restoring process takes more time, increasing the latency. Ways to reduce such context/restore latency include having relatively few registers in their central processing units (undesirable because it slows down most non-interrupt processing substantially), or at least having the hardware not save them all (this fails if the software then needs to compensate by saving the rest "manually"). Another technique involves spending silicon gates on "shadow registers": One or more duplicate registers used only by the interrupt software, perhaps supporting a dedicated stack.

Other factors affecting interrupt latency include:

• Cycles needed to complete current CPU activities. To minimize those costs, microcontrollers tend to have short pipelines (often three instructions or less), small write buffers, and ensure that longer instructions are continuable or restartable. RISC design principles ensure that most instructions take the same number of cycles, helping avoid the need for most such continuation/restart logic.
The length of any critical section that needs to be interrupted. Entry to a critical section restricts concurrent data structure access. When a data structure must be accessed by an interrupt handler, the critical section must block that interrupt. Accordingly, interrupt latency is increased by however long that interrupt is blocked. When there are hard external constraints on system latency, developers often need tools to measure interrupt latencies and track down which critical sections cause slowdowns.

- One common technique just blocks all interrupts for the duration of the critical section. This is easy to implement, but sometimes critical sections get uncomfortably long.
- A more complex technique just blocks the interrupts that may trigger access to that data structure. This is often based on interrupt priorities, which tend to not correspond well to the relevant system data structures. Accordingly, this technique is used mostly in very constrained environments.
- Processors may have hardware support for some critical sections. Examples include supporting atomic access to bits or bytes within a word, or other atomic access primitives like the LDREX/STREX exclusive access primitives introduced in the ARMv6 architecture.
- Interrupt nesting. Some microcontrollers allow higher priority interrupts to interrupt lower priority ones. This allows software to manage latency by giving time-critical interrupts higher priority (and thus lower and more predictable latency) than less-critical ones.
- Trigger rate. When interrupts occur back-to-back, microcontrollers may avoid an extra context save/restore cycle by a form of tail call optimization.

Lower end microcontrollers tend to support fewer interrupt latency controls than higher end ones.

**History**

The first single-chip microprocessor was the 4-bit Intel 4004 released in 1971, with the Intel 8008 and other more capable microprocessors becoming available over the next several years.

These however all required external chip(s) to implement a working system, raising total system cost, and making it impossible to economically computerize appliances.

The Smithsonian Institution says TI engineers Gary Boone and Michael Cochran succeeded in creating the first microcontroller in 1971. The result of their work was the TMS 1000, which went commercial in 1974. It combined read-only memory, read/write memory, processor and clock on one chip and was targeted at embedded systems.[7]

Partly in response to the existence of the single-chip TMS 1000,[8] Intel developed a computer system on a chip optimized for control applications, the Intel 8048, with
commercial parts first shipping in 1977. It combined RAM and ROM on the same chip. This chip would find its way into over one billion PC keyboards, and other numerous applications. At this time Intels President, Luke J. Valenter, stated that the (Microcontroller) was one of the most successful in the companies history, and expanded the division's budget over 25%.

Most microcontrollers at this time had two variants. One had an erasable EPROM program memory, which was significantly more expensive than the PROM variant which was only programmable once. Erasing the EPROM required exposure to ultraviolet light through a transparent quartz lid. One-time parts could be made in lower-cost opaque plastic packages.

In 1993, the introduction of EEPROM memory allowed microcontrollers (beginning with the Microchip PIC16x84) to be electrically erased quickly without an expensive package as required for EPROM, allowing both rapid prototyping, and In System Programming.

The same year, Atmel introduced the first microcontroller using Flash memory.

Other companies rapidly followed suit, with both memory types.

Cost has plummeted over time, with the cheapest 8-bit microcontrollers being available for under $0.25 in quantity (thousands) in 2009, and some 32-bit microcontrollers around $1 for similar quantities.

Nowadays microcontrollers are low cost and readily available for hobbyists, with large online communities around certain processors.

In the future, MRAM could potentially be used in microcontrollers as it has infinite endurance and its incremental semiconductor wafer process cost is relatively low.

**Microcontroller embedded memory technology**

Since the emergence of microcontrollers, many different memory technologies have been used. Almost all microcontrollers have at least two different kinds of memory, a non-volatile memory for storing firmware and a read-write memory for temporary data.

**Data**

From the earliest microcontrollers to today, six-transistor SRAM is almost always used as the read/write working memory, with a few more transistors per bit used in the register file. MRAM could potentially replace it as it is 4-10 times denser which would make it more cost effective.

In addition to the SRAM, some microcontrollers also have internal EEPROM for data storage; and even ones that do not have any (or not enough) are often connected to
external serial EEPROM chip (such as the BASIC Stamp) or external serial flash memory chip.

A few recent microcontrollers beginning in 2003 have "self-programmable" flash memory.[9]

**Firmware**

The earliest microcontrollers used mask ROM to store firmware. Later microcontrollers (such as the early versions of the Freescale 68HC11 and early PIC microcontrollers) had quartz windows that allowed ultraviolet light in to erase the EPROM.

The Microchip PIC16C84, introduced in 1993,[10] was the first microcontroller to use EEPROM to store firmware.

Also in 1993, Atmel introduced the first microcontroller using NOR Flash memory to store firmware.[9]

PSoC microcontrollers, introduced in 2002, store firmware in SONOS flash memory.

MRAM could potentially be used to store firmware.

**Selecting a PIC - what is available?**

So which PIC should you choose to start with? Some 10 years ago this question was easy to answer: the 16F84 (or, before that chip was available, the now discontinued 16c84). These were the only affordable flash PICs and hence THE hobbyist PICs. (EPROM based chips are either program-once or require the use of an UV-eraser.) You will still find lots of designs in electronics magazines and on the internet using these chips. The 16F84A is the slightly newer (cheaper and faster!) version.

Next came the 16F628: 18 pins like the 16F84, pin-compatible, but twice the FLASH, more RAM, more peripherals, and cheaper than the 16F84! One would expect the hobby community to make an instant switch, but that did not happen. There was (and still is) so much information on the internet about 16F84(A) that a lot of beginners still choose this chip to start with. After their start they make their own webpage, which only strengthens the position of the 16F84. In my opinion starting with a 16F84 is a bad choice on pure technical grounds, but if you found a nice introduction that features the 16F84 you might want to use it. Just be aware that you are trying to learn to drive a car on a Ford T.

Note that there are some that say that the 16F84 is still the best PIC to start with, because it is (slightly) easier to use than the later chips. This is mainly caused by the smaller code size (so code paging is not an issue) and by the lack of any analog peripherals (so there is no need to switch them off before the pins are used in digital mode). Personaly I think the
added complexity of a 16F628A, 16F88, 16F630, etc is a small price to pay for the lower price and extra features.

After the 16F628 Microchip has steadily widened its offering of flash chips with types that are ever more attractive. The 16F628 was replaced by the cheaper 16F628A, the 16F648A offers twice the code size and more RAM. The 40-pin 16F877 has much more of everything, and it has smaller 16F87x cousins with less pins or less memory if you don't need its full power. The 16F87x series was replaced by the 16F87xA series (cheaper, but beware that the programming algorithm is different). The 14-pins 16F630 and 16F676 appeared, offering a very good bang-for-your-bucks. The 8-pins 12F629 and 12F675 are nice for very small projects.

The above chips (16F and 12F) are all 14-bit core chips. This refers to the properties of the computing element (CPU) in the chip. If I remember correctly the 40-pin 18F452 chip was the first 16-bit core FLASH chip. This CPU is much easier to program, and can address larger amounts of code and data than the 14-bit CPU. Think of the 18F452 as a 16F877 with a redesigned engine.

The flood has not stopped: there are lots of 18F chips (40, 28 and 18 pins DIP, even more pins in various SMD housings), and a really huge variety of 12F/16F chips (40, 28, 18, 14 and 8 pins, with 20 pins coming soon). Microchip is also transferring its older 12-bit core chips to flash versions, check for instance the 12F509. The latest additions are the tiny 10F chips: these are very small 6-pin SMD chips, but they are also available in 8-pin DIP (with two unused pins).

Microchip has also expanded its product line into the territory of the DSPs (Digital Signal Processors) with the dsPIC30F family. These are in my opinion interesting chips, but they are so unlike the other PICs that they should be treated as an entirely different product line.

I recently found one reason to prefer the 16F628 over any other PIC: this chip has provides a clock option where one external resistor, which carries only DC, determines the clock frequency. A lot of other PICs provide the option to determine the clock frequency with an external resistor + capacitor, but in that case both components carry AC and practical capacitor values are very small so stray capacitance can have a big influence on the frequency. With the 16F628 it is very practical to let an external potentiometer determine the speed of the PIC, which can lead to a very simple circuit.

So which chip should you choose? If that has not been decided for you by some webpage or book you want to use, or by the development tools you like, or by what you local electronics shop has in stock, or what your wallet allows you, let's see what I can advise.

Your first impulse might be to take the cheapest chip that seems capable of what you want (the most obvious criteria are often the number of I/O pins and the availability of peripherals like an UART or and A/D converter). If you want to follow this path I would advise you to take a chip that is at least one step 'fatter' than your original choice. That
will give you some breathing room when your application expands, and some I/O pins to aid in debugging (so you could attach for instance an LCD). In fact I would advise you to take the fattest chip that still has the same architecture (CPU core), treating the 12/14 bit cores as the same. The extras will be very useful while debugging, and the extra cost is small. If and when your application is fully debugged you can transfer to the chip you originally had in mind. (Or just keep the fat chip. Why change a winning team?)

Some people take this reasoning to the extreme: forget all the small chips, forget the 12/14 bit cores, take a big 18F, or even a dsPIC. I agree to some extent, but if you want to do more than a few PIC projects this approach might get a little expensive, because the 18F and dsPIC families don't yet have the really small and cheap chips you can find in the 12/14 bit families.

So my advise is: first decide which family you want to use. If you want to do a large number of projects, look low (12/14 bit core). If you need the CPU horsepower, large memories, or other goodies look high (18F, or even dsPIC). Next take a fat DIP chip within that family. Within the 12/14 bit cores that would be the 16F877A (12/14 bit core). The developments in the 16 bit cores (18F) are too fast to follow. The 18F452 used to be the king here, but it has been replaced by the 18F4520, and even larger chips are available (18F4620). I am not sufficiently experienced with these chips to advice on dsPICs, but the 30F4013 is currently the fattest 40-pin dsPIC.

If you are relatively new to electronics you might kill a few chips in your learning process. It might be a good idea to let these chips be somewhat less expensive than the fat ones I advised you. For this purpose I suggest one of the 14-pins chips from the 14-bit core family. The lowly 16F630 is cheap, the 16F688 is the fattest chip in this group. The 8 and 6 pin chips are slightly cheaper, but have so few I/O pins that I would not recommend them as starters.

Ubicom sells the SX series of PIC clone. These chips are interesting because they provide MUCH more computing power than the Microchip 12/14 bit core PICs. On the downside the SX'es do not provide much peripherals (only a comparator and a timer), so you will need those MIPS to implement what the manufacturer call virtual peripherals. This is a nice and powerful concept, but IMHO not suited to a beginner. Note that the SX chips can run at one clock cycle per instruction (PICs need four clock cycles per instruction). So a 50 MHz SX is equivalent to a 200 MHz PIC. But an SX needs the equivalent of three instructions for when it changes its program counter (a PIC needs two), and the SX chips have a (somewhat enhanced) 12-bit core.

The table below compares some PICs that can be interesting.

<table>
<thead>
<tr>
<th>chip</th>
<th>package</th>
<th>I/O</th>
<th>code</th>
<th>data</th>
<th>EEPROM</th>
<th>peripheral</th>
<th>MIPS</th>
<th>US$</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>12C509</td>
<td>sdip 8</td>
<td>6</td>
<td>1k</td>
<td>41</td>
<td>-</td>
<td>osc</td>
<td>1</td>
<td>1.80</td>
<td>OTP</td>
</tr>
<tr>
<td>Code</td>
<td>Package</td>
<td>I/O</td>
<td>Cap</td>
<td>Speed</td>
<td>Features</td>
<td>Price</td>
<td>Notes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>------</td>
<td>-----</td>
<td>--------</td>
<td>----------</td>
<td>-------</td>
<td>----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12F629</td>
<td>sdip 8</td>
<td>6</td>
<td>1k</td>
<td>64</td>
<td>osc</td>
<td>5</td>
<td>1.60 cheap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12F675</td>
<td>sdip 8</td>
<td>6</td>
<td>1k</td>
<td>64</td>
<td>a/d, osc</td>
<td>5</td>
<td>1.90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16C84</td>
<td>sdip 18</td>
<td>13</td>
<td>1k</td>
<td>38</td>
<td>64</td>
<td>2.5</td>
<td>discontinued</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F84</td>
<td>sdip 18</td>
<td>13</td>
<td>1k</td>
<td>68</td>
<td>64</td>
<td>6.00</td>
<td>obsolete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F84A</td>
<td>sdip 18</td>
<td>13</td>
<td>1k</td>
<td>38</td>
<td>64</td>
<td>5</td>
<td>obsolete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F628</td>
<td>sdip 18</td>
<td>16</td>
<td>2k</td>
<td>224</td>
<td>d/a, uart, osc</td>
<td>5</td>
<td>3.50 3rd choice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F870</td>
<td>sdip 28</td>
<td>22</td>
<td>2k</td>
<td>128</td>
<td>64</td>
<td>5</td>
<td>5.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F871</td>
<td>wdip 40</td>
<td>33</td>
<td>2k</td>
<td>128</td>
<td>64</td>
<td>5</td>
<td>5.90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F872</td>
<td>sdip 28</td>
<td>22</td>
<td>2k</td>
<td>128</td>
<td>64</td>
<td>5</td>
<td>4.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F873</td>
<td>sdip 28</td>
<td>22</td>
<td>4k</td>
<td>192</td>
<td>128</td>
<td>5</td>
<td>7.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F874</td>
<td>wdip 40</td>
<td>33</td>
<td>4k</td>
<td>192</td>
<td>128</td>
<td>5</td>
<td>7.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F876</td>
<td>sdip 28</td>
<td>22</td>
<td>8k</td>
<td>368</td>
<td>256</td>
<td>5</td>
<td>8.20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16F877</td>
<td>wdip 40</td>
<td>33</td>
<td>8k</td>
<td>368</td>
<td>256</td>
<td>5</td>
<td>9.50 2nd choice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18F242</td>
<td>sdip 28</td>
<td>34</td>
<td>8k</td>
<td>512</td>
<td>256</td>
<td>5</td>
<td>8.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18F252</td>
<td>sdip 28</td>
<td>34</td>
<td>16k</td>
<td>153</td>
<td>256</td>
<td>10</td>
<td>9.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18F442</td>
<td>wdip 40</td>
<td>34</td>
<td>8k</td>
<td>512</td>
<td>256</td>
<td>10</td>
<td>9.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18F452</td>
<td>wdip 40</td>
<td>34</td>
<td>16k</td>
<td>153</td>
<td>256</td>
<td>10</td>
<td>10.0 1st choice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SX18</td>
<td>sdip 20</td>
<td>12</td>
<td>2k</td>
<td>136</td>
<td>-</td>
<td>50</td>
<td>discontinued</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SX28</td>
<td>sdip 28</td>
<td>20</td>
<td>2k</td>
<td>136</td>
<td>-</td>
<td>50</td>
<td>4.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SX48</td>
<td>TQFP 48</td>
<td>36</td>
<td>4k</td>
<td>262</td>
<td>-</td>
<td>50</td>
<td>7.40 no DIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SX52</td>
<td>PQFP 52</td>
<td>40</td>
<td>4k</td>
<td>262</td>
<td>-</td>
<td>50</td>
<td>7.40 no DIP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The price is of course an indication only, but you should be able to get a single chip for the indicated price (excluding taxes and shipping). Both the absolute and relative prices will vary between sources.
The 12C509 is still popular for hacking pay-TV or game consoles, but its role as small and cheap PIC has been taken over by the 12F629. Note that a 12C509 can be programmed only once (OTP), so you must use an expensive 12C509JW (and an EPROM eraser) for development.

The 12F629 and 12F675 are very cheap 8-pin chips, suitable for projects that do not need the larger amount of code space, data space, I/O pins, peripherals, etc. which are present on the larger (and more expensive) chips. But I do not recommend these chips for a beginner because the code space and I/O pins of the larger chips make debugging much easier.

The 16C84 was the first re-programmable PIC. It was (and still is) featured in many designs on web pages and in magazines. The 16C84 has long been superseded by the 16F84 and the 16F84A. Except for existing designs or to use existing documentation these chips should be avoided: the 16F628 offers more memory for a lower price, in the same pinout.

The 16F628 can be considered the next-generation 16F84, because it is pin-compatible with those older chips. But note that it is not fully software compatible. The 16F628 also has a smaller cousin, the 16F627 (1k code instead of 2k, not shown in the table). The 16F627 does not seem to be an attractive chip as the prices I found were actually a little higher than for the 16F628.

With 8K code space and 34 I/O pins the 16F877 is the largest chip of the 16F87x family. The 16F876 comes in a smaller package with less (IO) pins. It is about the same price as a 16F877, so it is interesting only when the larger package of the 16F877 is a problem. The 16F873 and 16F874 have less resources and use a more cumbersome RAM address mapping to be compatible with older (non-flash) PICs. The 16F870 and 16F871 have even less resources but use the same RAM addressing as the 16F877. Note that the 16F877 and 16F876 both have a UART (for asynchronous serial communication) and an MSSP (for SPI and I2C), while the smaller chips have only a UART. The 16F872 is a 16F870 but with an MSSP instead of a UART.

The 18F chips are new family of PICs, with an instruction set that is much improved over the 16F chips, with more peripherals, and more code and data space. Yet the price of the 18F chips is only marginally higher than the comparable 16F87x chips. There are variations of the 18F chips (not shown in the table) that have an integrated CAN controller - nice when you want to create a network of PIC chips.

If you can't make sense of the Microchip part numbering you are not the only one. These are the only patterns that I have found:

- The prefix **12** is for chips with 8 pins.
- The prefix **16** is for 12-bit and 14-bit core chips with more than 8 pins.
- The prefix **18** is for 16-bit core chips.
Next the letter C is for EPROM (OTP or windowed) chips, except for the 16C84 that has EEPROM, which is (for a user) almost the same as flash.

The letter F is for flash chips.

Windowed EPROM chips have a JW suffix.

For some of the chips mentioned in the table Microchip has released improved versions, identified by appending an A to the type. Such A chips are in most aspects identical to their non-A predecessors (but it does not harm to check the data sheets or the 'migration' document), except that the programming algorithm often changed. Hence you can buy and use an A chip if it is available (they are often slightly cheaper), but check that your programmer explicitly supports the A version. Note: The 16F84A uses the same programming algorithm as the 16F84, but the A chip can run at up to 20 MHz, the non-A only up to 10 MHz.

So what chip should you choose to start with? As said before, first check which chips you can actually buy. Then consider whether you want to use an existing design or other document. In that case the choice has been narrowed down for you. If you already have a programmer, check which chips it supports. For the choice I recommend that you take the most powerful chip that still fulfills the above constraints. The 18F452 (the largest chip of the 18F family) would be the first choice, the 16F877 (the largest chip of the 16F87x family) the next, and the 16F628 the last.

Once you have acquired some experience with your first PIC, and you have a nice project debugged and running, it might be the right time to fit it into a cheaper PIC.

**Clock options**

A PIC has a number of clock options. For most PICs the options are:

- HS: high-speed crystal (4 .. 20 MHz)
- XT: medium-speed crystal (200 kHz .. 4 MHz)
- LP: low-power 32768 Hz .. 200 kHz watch-style crystal
- RC: (external) capacitor + resistor

Instead of a crystal a (cheaper) ceramic resonator can be used. The HS, XT or LP options can also be used with an externally generated clock, connected to OSC1. Note that for a 4 MHz crystal either the XT or HS setting can be used.

When a crystal or resonator is used two capacitors are required, from each of the OSC pins to GND. The value depends on the frequency. I use 20 pF for 4, 10, and 20 MHz. 3-pin resonators have build-in capacitors. It is advised to keep the leads from these
capacitors to the GND pin short. (This makes me wonder why Microchip has on most
PICs placed the OSC pins next to the VCC, and the GND on the other side.)

Some PICs have other clock options:

- The 16F62x, 12Fxxx 12C509 provide an INTRC mode that generates an internal
clock of approximately 4 MHz. One or both of the pins that are normally used for the
crystal can be configured for IO. This is especially important on the 8-pin chips which
would - when an external crystal and reset were used - have only 4 IO pins left.
- The 16F628 has an ER mode where an external resistor and an internal capacitor
determine the clock. This mode has the big advantage over the RC mode that the
external resistor carries only a DC current, so long leads can be used (for instance to a
front-mounted potentiometer) without problems.
- The 18Fxxx chips provide a PLL setting which generates an internal clock of four
times the external (crystal controlled) clock. This can be used to get a 40 MHz
internal clock, with only a 10 MHz crystal (note: A 40 MHz crystal is not supported).

The internal and external RC clocks have an (in) accuracy of a few %. This is adequate
for flash-a-LED applications, but either not or just barely for more timing-critical things
like asynchronous serial communication.

When you build an existing design you have no choice but to follow its choice of
clocking, but for your own first experiment you could use a 'low cost' clock: INTRC for a
16F628 (4 MHz), RC for a 16F87x (100pF, 10k => 0.877 MHz). Note that the RC
characterization data (relation between RC values and frequency) for the 16F877 is not in
the 16F877 data sheet but in section 31.3.3 of the midrange reference manual. But
immediately after the 'first step' I recommend to use a crystal, and maybe use other clock
options later when this fits the design better. Using the 8-pin 12C509 and 12Fxxx with an
external crystal makes sense only when the remaining 4 IO pins are sufficient, and the
increased accuracy of a crystal over the internal oscillator is required (or the full speed of
20 MHz is needed).

1.4 Writing and compiling your program

The first step is to write our code. Every source file is saved in a single text file with
extension .pbas. Here is an example of one simple BASIC program, blink.pbas.

    program LED_Blink

    main:

    TRISB = 0           ' Configure pins of PORTB as output
    eloop:
        PORTB = $FF      ' Turn on diodes on PORTB
        Delay_ms(1000)   ' Wait 1 second
PORTB = 0  ; Turn off diodes on PORTB
Delay_ms(1000) ; Wait 1 second
goto eloop  ; Stay in loop
end.

When the program is completed and saved as .pbas file, it can be compiled by clicking on Compile Icon (or just hit CTRL+F9) in mikroBasic IDE. The compiling procedure takes place in two consecutive steps:

1. Compiler will convert .pbas file to assembly code and save it as blink.asm file.
2. Then, compiler automatically calls assembly, which converts .asm file into executable HEX code ready for feeding to microcontroller.

You cannot actually make the difference between the two steps, as the process is completely automated and indivisible. In case of syntax error in program code, program will not be compiled and HEX file will not be generated. Errors need to be corrected in the original .pbas file and then the source file may be compiled again. The best approach is to write and test small, logical parts of the program to make debugging easier.

1.5 Loading program to microcontroller

As a result of successful compiling of our previous code, mikroBasic will generate following files:

- blink.asm - assembly file
- blink.lst - program listing
- blink.mcl - mikro compile library
- blink.hex - executable file which is written into the programming memory

MCL file (mikro compile library) is created for each module you have included in the project. In the process of compiling, .mcl files will be linked together to output asm, lst and hex files. If you want to distribute your module without disclosing the source code, you can send your compiled library (file extension .mcl). User will be able to use your library as if he had the source code. Although the compiler is able to determine which routines are implemented in the library, it is a common practice to provide routine prototypes in a separate text file.

HEX file is the one you need to program the microcontroller. Commonly, generated HEX will be standard 8-bit Merged Intel HEX format, accepted by the vast majority of the programming software. The programming device (programmer) with accessory software installed on PC is in charge of writing the physical contents of HEX file into the internal memory of a microcontroller. The contents of a file blink.hex is given below:

:100000000428FF3FFF3FFF3F031383168601FF30A5
:1000100083128600630F000FF30F100FF30F2005E
:10002000F00B1328A28F10B16281928F20B1628A2
:10003000132810281A30F000FF30F100F00B2128AF
Beside loading a program code into programming memory, programmer also configures the target microcontroller, including the type of oscillator, protection of memory against reading, watchdog timer, etc. The following figure shows the connection between PC, programming device and the MCU.

Note that the programming software should be used only for the communication with the programming device — it is not suitable for code writing.

back to the top

1.6 Running the program

For proper functioning of microcontroller, it is necessary to provide power supply, oscillator, and a reset circuit. The supply can be set with the simple rectifier with Gretz junction and LM7805 circuit as shown in the figure below.
Oscillator can be 4MHz crystal and either two 22pF capacitors or the ceramic resonator of the same frequency (ceramic resonator already contains the mentioned capacitors, but unlike oscillator has three termination instead of only two). The rate at which the microcontroller operates, i.e. the speed at which the program runs, depends heavily on the oscillator frequency. During the application development, the easiest thing to do is to use the internal reset circuit — MCLR pin is connected to +5V through a 10K resistor. Below is the scheme of a rectifier with LM7805 circuit which gives the output of stable +5V, and the minimal configuration relevant for the operation of a PIC microcontroller.

After the supply is brought to the circuit previously shown, PIC microcontroller should look animated, and the LED diode should blink once every second. If the signal is completely missing (LED diode does not blink), then check if +5V is present at all the relevant pins of PIC.
ADC Library

ADC (Analog to Digital Converter) module is available with a number of PIC MCU models. Library function ADC_Read is included to provide you comfortable work with the module. The function is currently unsupported by the following PIC MCU models: P18F2331, P18F2431, P18F4331, and P18F4431.

5.2.2.1 ADC_Read – Get the results of AD conversion

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub function ADC_Read(dim Channel as byte) as word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Routine initializes ADC module to work with RC clock. Clock determines the time period necessary for performing AD conversion (min 12TAD). RC sources typically have Tad 4μS. Parameter &lt;Channel&gt; determines which channel will be sampled. Refer to the device data sheet for information on device channels.</td>
</tr>
<tr>
<td>Example</td>
<td>res = ADC_Read(2) ' reads channel 2 and stores value in variable res</td>
</tr>
</tbody>
</table>

ADC HW connection
EEPROM_Read – *Reads 1 byte from EEPROM*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub function Eeprom_Read(dim Address as byte) as byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Function reads byte from <code>&lt;Address&gt;</code>. <code>&lt;Address&gt;</code> is of byte type, which means it can address only 256 locations. For PIC18 MCU models with more EEPROM data locations, it is programmer's responsibility to set SFR EEADRH register appropriately. Ensure minimum 20ms delay between successive use of routines EEpROM_Write and EEpROM_Read. Although EEPROM will write the correct value, EEpROM_Read might return undefined result.</td>
</tr>
</tbody>
</table>
| Example         | TRISB = 0  
                 | Delay_ms(30)  
                 | for i = 0 to 20  
                 | PORTB = EEpROM_Read(i)  
                 | for j = 0 to 200  
                 |     Delay_us(500)  
                 |     next j  
                 | next i |

5.2.6.2 EEpROM_Write – *Writes 1 byte to EEPROM*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure EEprom_Write(dim Address as byte, dim Data as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Function writes byte to <code>&lt;Address&gt;</code>. <code>&lt;Address&gt;</code> is of byte type, which means it can address only 256 locations. For PIC18 MCU models with more EEPROM data locations, it is programmer's responsibility to set SFR EEADRH register appropriately. All interrupts will be disabled during execution of EEpROM_Write routine (GIE bit of INTCON register will be cleared). Routine will set this bit on exit. Ensure minimum 20ms delay between successive use of routines EEpROM_Write and EEpROM_Read. Although EEPROM will write the correct value, EEpROM_Read might return undefined result.</td>
</tr>
</tbody>
</table>
| Example         | for i = 0 to 20  
                 |     EEpROM_Write(i, i + 6)  
                 | next i |
5.2.7 Flash Memory Library

This library provides routines for accessing microcontroller Flash memory.

**Note:** Routines differ for PIC16 and PIC18 families.

### 5.2.7.1 Flash_Read – *Reads data from microcontroller Flash memory*

| Prototype | sub function Flash_Read(dim Address as longint) as byte ’ for PIC18  
|           | sub function Flash_Read(dim Address as word) as word ’ for PIC16 |
| Description | Procedure reads data from the specified <Address>. |
| Example | for i = 0 to 63  
|          | toRead = Flash_Read($0D00 + i)  
|          | ’ read 64 consecutive locations starting from 0x0D00  
|          | next i |

### 5.2.7.2 Flash_Write – *Writes data to microcontroller Flash memory*

| Prototype | sub procedure Flash_Write(dim Address as longint, dim byref Data as byte[64]) ’ for PIC18  
|           | sub procedure Flash_Write(dim Address as word, dim Data as word) ’ for PIC16 |
| Description | Procedure writes chunk of data to Flash memory (for PIC18, data needs to exactly 64 bytes in size). Keep in mind that this function erases target memory before writing <Data> to it. This means that if write was unsuccessful, your previous data will be lost. |
| Example | for i = 0 to 63  
|          | toWrite[i] = i  
|          | next i  
|          | Flash_Write($0D00, toWrite)  
|          | ’ write contents of the array to the address 0x0D00 |

### LCD Library

BASIC provides a set of library procedures and functions for communicating with commonly used 4-bit interface LCD (with Hitachi HD44780 controller). Be sure to designate port with LCD as output, before using any of the following library procedures or functions.
### 5.2.9.1 LCD_Init – Initializes LCD with default pin settings

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD_Init(dim byref Port as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Initializes LCD at <code>&lt;Port&gt;</code> with default pin settings (see the figure below).</td>
</tr>
<tr>
<td>Example</td>
<td>LCD_Init(PORTB)</td>
</tr>
<tr>
<td></td>
<td>' Initializes LCD on PORTB (check pin settings in the figure below)</td>
</tr>
</tbody>
</table>

### 5.2.9.2 LCD_Config – Initializes LCD with custom pin settings

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD_Config(dim byref Port as byte, const RS, const EN, const WR, const D7, const D6, const D5, const D4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Initializes LCD at <code>&lt;Port&gt;</code> with pin settings you specify: parameters <code>&lt;RS&gt;</code>, <code>&lt;EN&gt;</code>, <code>&lt;WR&gt;</code>, <code>&lt;D7&gt;</code> .. <code>&lt;D4&gt;</code> need to be a combination of values 0..7 (e.g. 3,6,0,7,2,1,4).</td>
</tr>
<tr>
<td>Example</td>
<td>LCD_Config(PORTD, 1, 2, 0, 3, 5, 4, 6)</td>
</tr>
<tr>
<td></td>
<td>' Initializes LCD on PORTD with our custom pin settings</td>
</tr>
</tbody>
</table>

### 5.2.9.3 LCD_Chr – Prints char on LCD at specified row and col

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD_Chr(dim Row as byte, dim Column as byte, dim Character as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Prints <code>&lt;Character&gt;</code> at specified <code>&lt;Row&gt;</code> and <code>&lt;Column&gt;</code> on LCD.</td>
</tr>
<tr>
<td>Example</td>
<td>LCD_Chr(1, 2, &quot;e&quot;)</td>
</tr>
<tr>
<td></td>
<td>' Prints character &quot;e&quot; on LCD (1st row, 2nd column)</td>
</tr>
</tbody>
</table>

### 5.2.9.4 LCD_Chr_CP – Prints char on LCD at current cursor position

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD_Chr_CP(dim Character as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Prints <code>&lt;Character&gt;</code> at current cursor position.</td>
</tr>
<tr>
<td>Example</td>
<td>LCD_Chr_CP(&quot;k&quot;)</td>
</tr>
<tr>
<td></td>
<td>' Prints character &quot;k&quot; at current cursor position</td>
</tr>
</tbody>
</table>

### 5.2.9.5 LCD_Out – Prints string on LCD at specified row and col

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD_Out(dim Row as byte, dim Column as byte, dim byref Text as char[255])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Prints <code>&lt;Text&gt;</code> (string variable) at specified <code>&lt;Row&gt;</code> and <code>&lt;Column&gt;</code> on LCD. Both string variables and string constants can be passed.</td>
</tr>
</tbody>
</table>
Example | LCD_Out(1, 3, Text)  
| ' Prints string variable Text on LCD (1st row, 3rd column)

### 5.2.9.6 LCD_Out_CP – Prints string on LCD at current cursor position

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD_Out_CP(dim byref Text as char[255])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Prints &lt;Text&gt; (string variable) at current cursor position. Both string variables and string constants can be passed.</td>
</tr>
</tbody>
</table>
| Example | LCD_Out_CP("Some text")  
| ' Prints "Some text" at current cursor position |

### 5.2.9.7 LCD_Cmd – Sends command to LCD

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD_Cmd(dim Command as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Sends &lt;Command&gt; to LCD.</td>
</tr>
</tbody>
</table>

List of available commands follows:

- **LCD_First_Row**  
  ' Moves cursor to 1st row

- **LCD_Second_Row**  
  ' Moves cursor to 2nd row

- **LCD_Third_Row**  
  ' Moves cursor to 3rd row

- **LCD_Fourth_Row**  
  ' Moves cursor to 4th row

- **LCD_Clear**  
  ' Clears display

- **LCD_Return_Home**  
  ' Returns cursor to home position,  
  ' returns a shifted display to original position.  
  ' Display data RAM is unaffected.

- **LCD_Cursor_Off**  
  ' Turn off cursor

- **LCD_Underline_On**  
  ' Underline cursor on

- **LCD_Blink_Cursor_On**  
  ' Blink cursor on

- **LCD_Move_Cursor_Left**
' Move cursor left without changing display data RAM

LCD_Move_Cursor_Right
' Move cursor right without changing display data RAM

LCD_Turn_On
' Turn LCD display on

LCD_Turn_Off
' Turn LCD display off

LCD_Shift_Left
' Shift display left without changing display data RAM

LCD_Shift_Right
' Shift display right without changing display data RAM

Example
LCD_Cmd(LCD_Clear)  ' Clears LCD display

---

5.2.10 LCD8 Library (8-bit interface LCD)

BASIC provides a set of library procedures and functions for communicating with commonly used 8-bit interface LCD (with Hitachi HD44780 controller). Be sure to designate Control and Data ports with LCD as output, before using any of the following library procedures or functions.
### 5.2.10.1 LCD8_Init – Initializes LCD with default pin settings

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD8_Init(dim byref Port_Ctrl as byte, dim byref Port_Data as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Initializes LCD at &lt;Port_Ctrl&gt; and &lt;Port_Data&gt; with default pin settings (see the figure below).</td>
</tr>
</tbody>
</table>
| Example | LCD8_Init(PORTB, PORTC)  
' Initializes LCD on PORTB and PORTC with default pin settings  
' (check pin settings in the figure below) |

### 5.2.10.2 LCD8_Config – Initializes LCD with custom pin settings

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD8_Config(dim byref Port_Ctrl as byte, dim byref Port_Data as byte, const RS, const EN, const WR, const D7, const D6, const D5, const D4, const D3, const D2, const D1, const D0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Initializes LCD at &lt;Port_Ctrl&gt; and &lt;Port_Data&gt; with pin settings you specify: parameters &lt;RS&gt;, &lt;EN&gt;, &lt;WR&gt; need to be in range 0..7; parameters &lt;D7&gt;..&lt;D0&gt; need to be a combination of values 0..7 (e.g. 3,6,5,0,7,2,1,4).</td>
</tr>
</tbody>
</table>
| Example | LCD8_Config(PORTC, PORTD, 0, 1, 2, 6, 5, 4, 3, 7, 1, 2, 0)  
' Initializes LCD on PORTC and PORTD with our custom pin settings |

### 5.2.10.3 LCD8_Chr – Prints char on LCD at specified row and col

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD8_Chr(dim Row as byte, dim Column as byte, dim Character as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Prints &lt;Character&gt; at specified &lt;Row&gt; and &lt;Column&gt; on LCD.</td>
</tr>
</tbody>
</table>
| Example | LCD8_Chr(1, 2, "e")  
' Prints character "e" on LCD (1st row, 2nd column) |

### 5.2.10.4 LCD8_Chr_CP – Prints char on LCD at current cursor position

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure LCD8_Chr_CP(dim Character as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Prints &lt;Character&gt; at current cursor position.</td>
</tr>
</tbody>
</table>
| Example | LCD8_Chr_CP("k")  
' Prints character "k" at current cursor position |
5.2.10.5 LCD8_Out – *Prints string on LCD at specified row and col*

<table>
<thead>
<tr>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sub procedure</strong> LCD8_Out(dim Row as byte, dim Column as byte, dim byref Text as char[255])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prints <code>&lt;Text&gt;</code> (string variable) at specified <code>&lt;Row&gt;</code> and <code>&lt;Column&gt;</code> on LCD. Both string variables and string constants can be passed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| LCD8_Out(1, 3, Text)  
  ' Prints string variable Text on LCD (1st row, 3rd column) |

5.2.10.6 LCD8_Out_CP – *Prints string on LCD at current cursor position*

<table>
<thead>
<tr>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sub procedure</strong> LCD8_Out_CP(dim byref Text as char[255])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prints <code>&lt;Text&gt;</code> (string variable) at current cursor position. Both string variables and string constants can be passed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| LCD8_Out_CP("Test")  
  ' Prints "Test" at current cursor position |

5.2.10.7 LCD8_Cmd – *Sends command to LCD*

<table>
<thead>
<tr>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sub procedure</strong> LCD8_Cmd(dim Command as byte)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sends <code>&lt;Command&gt;</code> to LCD.</td>
</tr>
</tbody>
</table>

List of available commands follows:

- **LCD_First_Row**
  ' Moves cursor to 1st row

- **LCD_Second_Row**
  ' Moves cursor to 2nd row

- **LCD_Third_Row**
  ' Moves cursor to 3rd row

- **LCD_Fourth_Row**
  ' Moves cursor to 4th row

- **LCD_Clear**
  ' Clears display

- **LCD_Return_Home**
  ' Returns cursor to home position,  
  ' returns a shifted display to original position.  
  ' Display data RAM is unaffected.

- **LCD_Cursor_Off**
' Turn off cursor

LCD_Underline_On
' Underline cursor on

LCD_Blink_Cursor_On
' Blink cursor on

LCD_Move_Cursor_Left
' Move cursor left without changing display data RAM

LCD_Move_Cursor_Right
' Move cursor right without changing display data RAM

LCD_Turn_On
' Turn LCD display on

LCD_Turn_Off
' Turn LCD display off

LCD_Shift_Left
' Shift display left without changing display data RAM

LCD_Shift_Right
' Shift display right without changing display data RAM

**Example**

LCD8_Cmd(LCD_Clear)    ' Clears LCD display

---

**PIC MCU**

**Pins of any port (with 8 pins)**

- PIN0
- PIN1
- PIN2
- PIN3
- PIN4
- PIN5
- PIN6
- PIN7

---

**PIC**

- PIN7
- PIN6
- PIN5
- PIN4
- PIN3
- PIN2
- PIN1
- PIN0

---

**LCD**

- D7
- D6
- D5
- D4
- E
- RS

---

**MikroElektronika**

---

**LCD HW connection**
5.2.11 Graphic LCD Library

mikroPascal provides a set of library procedures and functions for drawing and writing on Graphical LCD. Also it is possible to convert bitmap (use menu option Tools > BMP2LCD) to constant array and display it on GLCD. These routines works with commonly used GLCD 128x64, and work only with the PIC18 family.

5.2.11.1 GLCD_Config – Initializes GLCD with custom pin settings

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Config(dim byref Ctrl_Port as byte, dim byref Data_Port as byte, dim Reset as byte, dim Enable as byte, dim Rs as byte, dim Rw as byte, dim Cs1 as byte, dim Cs2 as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Initializes GLCD at &lt;Ctrl_Port&gt; and &lt;Data_Port&gt; with custom pin settings.</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_LCD_Config(PORTB, PORTC, 1, 7, 4, 6, 0, 2)</td>
</tr>
</tbody>
</table>

5.2.11.2 GLCD_Init – Initializes GLCD with default pin settings

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Init(dim Ctrl_Port as byte, dim Data_Port as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Initializes LCD at &lt;Ctrl_Port&gt; and &lt;Data_Port&gt;. With default pin settings Reset=7, Enable=1, RS=3, RW=5, CS1=2, CS2=0.</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_LCD_Init(PORTB, PORTC)</td>
</tr>
</tbody>
</table>

5.2.11.3 GLCD_Put_Ins – Sends instruction to GLCD.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Put_Ins(dim Ins as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Sends instruction &lt;Ins&gt; to GLCD. Available instructions include: X_ADRESS = $88 ' Adress base for Page 0 \nY_ADRESS = $40 ' Adress base for Y0 \nSTART_LINE = $C0 ' Adress base for line 0 \nDISPLAY_ON = $3F ' Turn display on \nDISPLAY_OFF = $3E ' Turn display off</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Put_Ins(DISPLAY_ON)</td>
</tr>
</tbody>
</table>

5.2.11.4 GLCD_Put_Data – Sends data byte to GLCD.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Put_Data(dim data as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Sends data byte to GLCD.</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>GLCD_Put_Data(temperature)</td>
</tr>
</tbody>
</table>

### 5.2.11.5 GLCD_Put_Data2 – Sends data byte to GLCD.

<table>
<thead>
<tr>
<th><strong>Prototype</strong></th>
<th>sub procedure GLCD_Put_Data2(dim data as byte, dim side as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Sends data to GLCD at specified <code>&lt;side&gt;</code> (<code>&lt;side&gt;</code> can take constant value LEFT or RIGHT).</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>GLCD_Put_Data2(temperature, 1)</td>
</tr>
</tbody>
</table>

### 5.2.11.6 GLCD_Select_Side- Selects the side of the GLCD.

<table>
<thead>
<tr>
<th><strong>Prototype</strong></th>
<th>sub procedure GLCD_Select_Side(dim LCDSide as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Selects the side of the GLCD: <code>const RIGHT = 0</code> <code>const LEFT = 1</code></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>GLCD_Select_Side(1)</td>
</tr>
</tbody>
</table>

### 5.2.11.7 GLCD_Data_Read – Reads data from GLCD.

<table>
<thead>
<tr>
<th><strong>Prototype</strong></th>
<th>sub function GLCD_Data_Read as byte</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Reads data from GLCD.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>GLCD_Data_Read</td>
</tr>
</tbody>
</table>

### 5.2.11.8 GLCD_Clear_Dot – Clears a dot on the GLCD.

<table>
<thead>
<tr>
<th><strong>Prototype</strong></th>
<th>sub procedure GLCD_Clear_Dot(dim x as byte, dim y as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Clears a dot on the GLCD at specified coordinates.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>GLCD_Clear_Dot(20, 32)</td>
</tr>
</tbody>
</table>

### 5.2.11.9 GLCD_Set_Dot – Draws a dot on the GLCD.

<table>
<thead>
<tr>
<th><strong>Prototype</strong></th>
<th>sub procedure GLCD_Set_Dot(dim x as byte, dim y as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Draws a dot on the GLCD at specified coordinates.</td>
</tr>
</tbody>
</table>
5.2.11.10 GLCD_Circle – *Draws a circle on the GLCD.*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Circle(dim CenterX as integer, dim CenterY as integer, dim Radius as integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Draws a circle on the GLCD, centered at (&lt;CenterX, CenterY&gt;) with (&lt;Radius&gt;).</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Circle(30, 42, 6)</td>
</tr>
</tbody>
</table>

5.2.11.11 GLCD_Line – *Draws a line*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Line(dim x1 as integer, dim y1 as integer, dim x2 as integer, dim y2 as integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Draws a line from ((x1,y1)) to ((x2,y2)).</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Line(0, 0, 120, 50)</td>
</tr>
</tbody>
</table>

5.2.11.12 GLCD_Invert – *Inverts display*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Invert(dim Xaxis as byte, dim Yaxis as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Procedure inverts display (changes dot state on/off) in the specified area, X pixels wide starting from 0 position, 8 pixels high. Parameter Xaxis spans 0..127, parameter Yaxis spans 0..7 (8 text lines).</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Invert(60, 6)</td>
</tr>
</tbody>
</table>

5.2.11.13 GLCD_Goto_XY – *Sets cursor to dot(x,y)*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Goto_XY(dim x as byte, dim y as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Sets cursor to dot ((x,y)). Procedure is used in combination with GLCD_Put_Data, GLCD_Put_Data2, and GLCD_Put_Char.</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Goto_XY(60, 6)</td>
</tr>
</tbody>
</table>

5.2.11.14 GLCD_Put_Char – *Prints \(<Character>\) at cursor position*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Put_Char(dim Character as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Prints <code>&lt;Character&gt;</code> at cursor position.</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Put_Char(k)</td>
</tr>
</tbody>
</table>

### 5.2.11.15 GLCD_Clear_Screen – *Clears the GLCD screen*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Clear_Screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Clears the GLCD screen.</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Clear_Screen</td>
</tr>
</tbody>
</table>

### 5.2.11.16 GLCD_Put_Text – *Prints text at specified position*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Put_Text(dim x_pos as word, dim y_pos as word, dim byref text as char[25], dim invert as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Prints <code>&lt;text&gt;</code> at specified position; y_pos spans 0..7.</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Put_Text(0, 7, My_text, NONINVERTED_TEXT)</td>
</tr>
</tbody>
</table>

### 5.2.11.17 GLCD_Rectangle – *Draws a rectangle*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Rectangle(dim X1 as byte, dim Y1 as byte, dim X2 as byte, dim Y2 as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Draws a rectangle on the GLCD. (x1,y1) sets the upper left corner, (x2,y2) sets the lower right corner.</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Rectangle(10, 0, 30, 35)</td>
</tr>
</tbody>
</table>

### 5.2.11.18 GLCD_Set_Font – *Sets font for GLCD*

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure GLCD_Set_Font(dim font_index as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Sets font for GLCD. Parameter <code>&lt;font_index&gt;</code> spans from 1 to 4, and determines which font will be used:</td>
</tr>
<tr>
<td></td>
<td>1: 5x8 dots</td>
</tr>
<tr>
<td></td>
<td>2: 5x7</td>
</tr>
<tr>
<td></td>
<td>3: 3x6</td>
</tr>
<tr>
<td></td>
<td>4: 8x8</td>
</tr>
<tr>
<td>Example</td>
<td>GLCD_Set_Font(2)</td>
</tr>
</tbody>
</table>
PWM Library

CCP (Capture/Compare/PWM) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide comfortable work with PWM (Pulse Width Modulation).

Note that these routines support module on PORTC pin RC2, and won't work with modules on other ports. Also, BASIC doesn't support enhanced PWM modules.

5.2.13.1 PWM_Init – Initializes PWM module

**Prototype**

\[
\text{sub procedure PWM_Init(const PWM_Freq)}
\]

**Description**

Initializes PWM module with (duty ratio) 0%. \(<PWM\_Freq>\) is a desired PWM frequency (refer to device data sheet for correct values in respect with Fosc).

**Example**

\[
\text{PWM_Init(5000) \ ' initializes PWM module, freq = 5kHz}
\]

5.2.13.2 PWM_Change_Duty – Changes duty ratio

**Prototype**

\[
\text{sub procedure PWM_Change_Duty(dim \ New\_Duty as byte)}
\]

**Description**

Routine changes duty ratio. \(<\text{New\_Duty}>\) takes values from 0 to 255, where 0 is 0% duty ratio, 127 is 50% duty ratio, and 255 is 100% duty ratio. Other values for specific duty ratio can be calculated as \((\text{Percent} \times 255)/100\).

**Example**

\[
\text{while true}
\]
\[
\text{Delay\_ms(100)}
\]
\[
\text{j = j + 1}
\]
\[
\text{PWM\_Change\_Duty(j)}
\]
\[
\text{wend}
\]

5.2.13.3 PWM_Start – Starts PWM

**Prototype**

\[
\text{sub procedure PWM\_Start}
\]

**Description**

Starts PWM.

**Example**

\[
\text{PWM\_Start}
\]

5.2.13.4 PWM_Stop – Stops PWM

**Prototype**

\[
\text{sub procedure PWM\_Stop}
\]
### USART Library

USART (Universal Synchronous Asynchronous Receiver Transmitter) hardware module is available with a number of PIC MCU models. You can easily communicate with other devices via RS232 protocol (for example with PC, see the figure at the end of this chapter - RS232 HW connection). You need a PIC MCU with hardware integrated USART (for example, PIC16F877). Then, simply use the functions and procedures described below.

**Note:** Some PIC micros that have two USART modules, such as P18F8520, require you to specify the module you want to use. Simply append the number 1 or 2 to procedure or function name, e.g. `USART_Write2(Dat)`.

#### 5.2.16.1 USART_Init – Initializes USART

<table>
<thead>
<tr>
<th><strong>Prototype</strong></th>
<th><code>sub procedure USART_Init(const Baud_Rate)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Initializes PIC MCU USART hardware and establishes communication at</td>
</tr>
</tbody>
</table>
specified <Baud_Rate>. Refer to the device data sheet for baud rates allowed for specific Fosc. If you specify the unsupported baud rate, compiler will report an error.

Example

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>USART_Init(2400)</td>
</tr>
</tbody>
</table>

5.2.16.2 USART_Data_Rdy – Checks if data is ready

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub function USART_Data_Rdy as byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Function checks if data is ready. Returns 1 if so, returns 0 otherwise.</td>
</tr>
<tr>
<td>Example</td>
<td>USART_Data_Rdy</td>
</tr>
</tbody>
</table>

5.2.16.3 USART_Read – Receives a byte

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub function USART_Read as byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Receives a byte; if byte is not received returns 0.</td>
</tr>
<tr>
<td>Example</td>
<td>USART_Read</td>
</tr>
</tbody>
</table>

5.2.16.4 USART_Write – Transmits a byte

<table>
<thead>
<tr>
<th>Prototype</th>
<th>sub procedure USART_Write(dim Data as byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Procedure transmits byte &lt;Data&gt;.</td>
</tr>
<tr>
<td>Example</td>
<td>USART_Write(dat)</td>
</tr>
</tbody>
</table>
**Introduction**

It is commonly said that microcontroller is an “entire computer on a single chip”, which implies that it has more to offer than a single CPU (microprocessor). This additional functionality is actually located in microcontroller’s subsystems, also called the “integrated peripherals”. These (sub)devices basically have two major roles: they expand the possibilities of the MCU making it more versatile, and they take off the burden for some repetitive and “dumber” tasks (mainly communication) from the CPU.

Every microcontroller is supplied with at least a couple of integrated peripherals – commonly, these include timers, interrupt mechanisms and AD converters. More powerful microcontrollers can command a larger number of more diverse peripherals. In this chapter, we will cover some common systems and the ways to utilize them from BASIC programming language.

**6.1 Interrupt Mechanism**

Interrupts are mechanisms which enable instant response to events such as counter overflow, pin change, data received, etc. In normal mode, microcontroller executes the
main program as long as there are no occurrences that would cause an interrupt. Upon interrupt, microcontroller stops the execution of main program and commences the special part of the program which will analyze and handle the interrupt. This part of program is known as the interrupt (service) routine.

In BASIC, interrupt service routine is defined by procedure with reserved name interrupt. Whatever code is stored in that procedure, it will be executed upon interrupt.

First, we need to determine which event caused the interrupt, as PIC microcontroller calls the same interrupt routine regardless of the trigger. After that comes the interrupt handling, which is executing the appropriate code for the trigger event.

Here is a simple example:

In the main loop, program keeps LED_run diode on and LED_int diode off. Pressing the button T causes the interrupt – microcontroller stops executing the main program and starts the interrupt procedure.

```
program testinterrupt

symbol LED_run = PORTB.7 ' LED_run is connected to PORTB pin 7
symbol LED_int = PORTB.6 ' LED_int is connected to PORTB pin 6

sub procedure interrupt ' Interrupt service routine
    if INTCON.RBIF = 1 then
        INTCON.RBIF = 0
        ' Changes on RB4-RB7 ?
    else if INTCON.INTF = 1 then
        ' External interrupt (RB0 pin)
```

Pressing the button T causes the interrupt INT.
LED_run = 0
LED_int = 1
Delay_ms(500)
INTCON.INTF = 0

else if INTCON.T0IF = 1 then  ' TMR0 interrupt occurred ?
    INTCON.T0IF = 0
    else if INTCON.EEIF = 1 then  ' Is EEPROM write cycle finished ?
        INTCON.EEIF = 0
    end if
end if
end if
end if
end sub

main:

TRISB = %00111111  ' Pins RB6 and RB7 are output
OPTION_REG = %10000000  ' Turn off pull-up resistors
                    ' and set interrupt on falling edge
                    ' of RB0 signal
INTCON = %10010000  ' Enable external interrupts
PORTB = 0  ' Initial value on PORTB

elool:  ' While there is no interrupt, program runs
in endless loop:
    LED_run = 1  ' LED_run is on
    LED_int = 0  ' LED_int is off
goto eloop

d. 

Now, what happens when we push the button? Our interrupt routine first analyzes the interrupt by checking flag bits with couple of if..then instructions, because there are several possible interrupt causes. In our case, an external interrupt took place (pin RB0/INT state changes) and therefore bit INTF in INTCON register is set. Microcontroller will change LED states, and provide a half second delay for us to actually see the change. Then it will clear INTF bit in order to enable interrupts again, and return to executing the main program.

In situations where microcontroller must respond to events unrelated to the main program, it is very useful to have an interrupt service routine. Perhaps, one of the best examples is multiplexing the seven-segment display – if multiplexing code is tied to timer interrupt, main program will be much less burdened because display refreshes in the background.
6.2 Internal AD Converter

A number of microcontrollers have built in Analog to Digital Converter (ADC). Commonly, these AD converters have 8-bit or 10-bit resolution allowing them voltage sensitivity of 19.5mV or 4.8mV, respectively (assuming that default 5V voltage is used).

The simplest AD conversion program would use 8-bit resolution and 5V of microcontroller power as referent voltage (value which the value "read" from the microcontroller pin is compared to). In the following example we measure voltage on RA0 pin which is connected to the potentiometer (see the figure below).

```
program ADC_8
main:
```

Potentiometer gives 0V in one terminal position and 5V in the other – since we use 8-bit conversion, our digitalized voltage can have 256 steps. The following program reads voltage on RA0 pin and displays it on port B diodes. If not one diode is on, result is zero and if all of diodes are on, result is 255.

```
program ADC_8
main:
```
TRISA = %1111111 ' Port A is input
PORTD = 0
TRISD = %00000000

ADCON1 = %1000010 ' Port A is in analog mode,
    ' 0 and 5V are referent voltage values,
    ' and the result is aligned right
    ' (higher 6 bits of ADRESH are zero).

ADCON0 = %11010001 ' ADC clock is generated by internal RC
    ' circuit; voltage is measured on RA2 and
    ' allows the use of AD converter

Delay_ms (500) ' 500 ms pause

eloop:
    ADCON0.2 = 1 ' Conversion starts

    wait:

        ' wait for ADC to finish
    Delay_ms(5)
    if ADCON0.2 = 1 then
goto wait
    end if

    PORTD = ADRESH ' Set lower 8 bits on port D
    Delay_ms(500) ' 500 ms pause
    goto eloop ' Repeat all
end loop. ' End of program.

First, we need to properly initialize registers ADCON1 and ADCON0. After that, we set
ADCON0.2 bit which initializes the conversion and then check ADCON0.2 to determine
if conversion is over. If over, the result is stored into ADRESH and ADRESL where from
it can be copied.

Former example could also be carried out via ADC_Read instruction. Our following
example uses 10-bit resolution:

program ADC_10

    dim AD_Res as word

main:
    TRISA  = %11111111 ' PORTA is input
TRISD  = %00000000 ' PORTD is output
ADCON1 = %1000010 ' PORTA is in analog mode,
    ' 0 and 5V are referent voltage values,
    ' and the result is aligned right

    eloop:
As one port is insufficient, we can use LCD for displaying all 10 bits of result. Connection scheme is below and the appropriate program follows. For more information on LCD routines, check Chapter 5.2: Library Routines.

```
program ADC_on_LCD

dim AD_Res as word
dim dummyCh as char[6]

main:

TRISA = %1111111  ' PORTA is input
TRISB = 0        ' PORTB is output (for LCD)

ADCON1 = %10000010    ' PORTA is in analog mode,
                      ' 0 and 5V are referent voltage values,
                      ' and the result is aligned right.

Lcd_Init(PORTB)   ' Initialize LCD
```
Lcd_Cmd(LCD_CLEAR)  ' Clear LCD
Lcd_Cmd(LCD_CURSOR_OFF)  ' and turn the cursor off

eloop:

AD_Res = ADC_Read(2)  ' Execute conversion and store result
   to variable AD_Res
LCD_Out(1, 1, " ")  ' Clear LCD from previous result
WordToStr(AD_Res, dummyCh)  ' Convert the result in text,
   and print it in line 1, char 1

Delay_ms(500)  ' 500 ms pause
goto eloop
end.  ' End of program

back to the top

6.3 TMR0 Timer

TMR0 timer is an 8-bit special function register with working range of 256. Assuming
that 4MHz oscillator is used, TMR0 can measure 0-255 microseconds range (at 4MHz,
TMR0 increments by one microsecond). This period can be increased if prescaler is used.
Prescaler divides clock in a certain ratio (prescaler settings are made in OPTION_REG
register).

Our following program example shows how to generate 1 second using TMR0 timer. For
visual purposes, program toggles LEDs on PORTB every second.

Before the main program, TMR0 should have interrupt enabled (bit 2) and GIE bit (bit 7)
in INTCON register should be set. This will enable global interrupts.

program Timer0_1sec
dim cnt as byte
dim a as byte
dim b as byte
sub procedure interrupt
   cnt = cnt + 1  ' Increment value of cnt on every interrupt
   TMR0 = 96  ' Set T0IE, clear T0IF
   INTCON = $20
end sub

main:
   a = 0
   b = 1
   OPTION_REG = $84  ' Assign prescaler to TMR0
   TRISB = 0  ' PORTB as output
   PORTB = $FF  ' Initialize PORTB
   cnt = 0  ' Initialize cnt
TMRO   =  96
INTCON = $A0          ' Enable TMRO interrupt

' If cnt is 200, then toggle PORTB LEDs and reset cnt
do
  if cnt = 200 then
    PORTB  =  not(PORTB)
    cnt = 0
  end if
loop until 0 = 1
end.

Prescaler is set to 32, so that internal clock is divided by 32 and TMR0 increments every 31 microseconds. If TMR0 is initialized at 96, overflow occurs in (256-96)*31 us = 5 ms. We increase cnt every time interrupt takes place, effectively measuring time according to the value of this variable. When cnt reaches 200, time will total 200*5 ms = 1 second.

6.4 TMR1 Timer

TMR1 timer is a 16-bit special function register with working range of 65536. Assuming that 4MHz oscillator is used, TMR1 can measure 0-65535 microseconds range (at 4MHz, TMR1 increments by one microsecond). This period can be increased if prescaler is used. Prescaler divides clock in a certain ratio (prescaler settings are made in T1CON register).

Before the main program, TMR1 should be enabled by setting the zero bit in T1CON register. First bit of the register defines the internal clock for TMR1 – we set it to zero. Other important registers for working with TMR1 are PIR1 and PIE1. The first contains overflow flag (zero bit) and the other is used to enable TMR1 interrupt (zero bit). With TMR1 interrupt enabled and its flag cleared, we only need to enable global interrupts and peripheral interrupts in the IINTCON register (bits 7 and 6, respectively).

Our following program example shows how to generate 10 seconds using TMR1 timer. For visual purposes, program toggles LEDs on PORTB every 10 seconds.

program Timer1_10sec

dim cnt as byte

sub procedure interrupt
  cnt = cnt + 1
  Pir1.0 = 0           ’ Clear TMR1IF
end sub

main:

TRISB = 0
T1CON = 1
PIR1.TMR1IF = 0          ’ Clear TMR1IF
PIE1 = 1
PORTB = $F0
cnt = 0
INTCON = $C0

' Enable interrupts
' Initialize cnt

' If cnt is 152, then toggle PORTB LEDs and reset cnt
do
    if cnt = 152 then
        PORTB = not(PORTB)
        cnt = 0
    end if
loop until 0 = 1
end.

Prescaler is set to 00 so there is no dividing the internal clock and overflow occurs every 65.536 ms. We increase cnt every time interrupt takes place, effectively measuring time according to the value of this variable. When cnt reaches 152, time will total 152*65.536 ms = 9.96 seconds.

back to the top

6.5 PWM Module

Microcontrollers of PIC16F87X series have one or two built-in PWM outputs (40-pin casing allows 2, 28-pin casing allows 1). PWM outputs are located on RC1 and RC2 pins (40-pin MCUs), or on RC2 pin (28-pin MCUs). Refer to PWM library (Chapter 5.2: Library Routines) for more information.

The following example uses PWM library for getting various light intensities on LED connected to RC2 pin. Variable which represents the ratio of on to off signals is continually increased in the loop, taking values from 0 to 255. This results in continual intensifying of light on LED diode. After value of 255 has been reached, process begins anew.

program PWM_LED_Test
dim j as byte
main:

TRISB = 0
PORTB = 0
j = 0
TRISC = 0
PORTC = $FF
PWM_Init(5000)
PWM_Start

while true
    Delay_ms(10)
    j = j + 1
end.
PWM Change Duty(j) ' Set new duty ratio
PORTB = CCPR1L ' Send value of CCPR1L to PORTB
wend
end.

6.6 Hardware UART module (RS-232 Communication)

The easiest way to transfer data between microcontroller and some other device, e.g. PC or other microcontroller, is the RS-232 communication (also referred to as EIA RS-232C or V.24). RS232 is a standard for serial binary data interchange between a DTE (Data terminal equipment) and a DCE (Data communication equipment), commonly used in personal computer serial ports. It is a serial asynchronous 2-line (Tx for transmitting and Rx for receiving) communication with effective range of 10 meters.

Microcontroller can establish communication with serial RS-232 line via hardware UART (Universal Asynchronous Receiver Transmitter) which is an integral part of PIC16F87X microcontrollers. UART contains special buffer registers for receiving and transmitting data as well as a Baud Rate generator for setting the transfer rate.

This example shows data transfer between the microcontroller and PC connected by RS-232 line interface MAX232 which has role of adjusting signal levels on the microcontroller side (it converts RS-232 voltage levels +/- 10V to TTL levels 0-5V and vice versa).
Our following program example illustrates use of hardware serial communication. Data received from PC is stored into variable `dat` and sent back to PC as confirmation of successful transfer. Thus, it is easy to check if communication works properly. Transfer format is 8N1 and transfer rate is 2400 baud.

```
program USART_Echo

dim dat as byte

main:

    USART_Init(2400)  ' Initialize USART module
    while true
        if USART_Data_Ready = 1 then
            dat = USART_Read  ' Read the received data
            USART_Write(dat)  ' Send data via USART
        end if
    wend

end.
```

In order to establish the communication, PC must have a communication software installed. One such communication terminal is part of mikroBasic IDE. It can be accessed by clicking Tools > Terminal from the drop-down menu. Terminal allows you to monitor transfer and to set all the necessary transfer settings. First of all, we need to set the transfer rate to 2400 to match the microcontroller's rate. Then, select the appropriate communication port by clicking one of the 4 available (check where you plugged the serial cable).

After making these adjustments, clicking Connect starts the communication. Type your message and click Send Message – message will be sent to the microcontroller and back, where it will be displayed on the screen.

Note that serial communication can also be software based on any of 2 microcontroller pins – for more information, check the Chapter 9: Communications.

**Examples with Displaying Data**

- Introduction
- 7.1 LED Diode
- 7.2 Seven-Segment Display
- 7.3 LCD Display, 4-bit and 8-bit Interface
- 7.4 Graphical LCD
- 7.5 Sound Signalization
**Introduction**

Microcontrollers deal very well with 0’s and 1’s, but humans do not. We need indicator lights, numbers, letters, charts, beepers… In order to comprehend the information presented quicker and better, we need that information to be displayed to us in many different ways. In practice, human - machine communication can require substantial (machine) resources, so it is sometimes better to dedicate an entire microcontroller to that task. This device is then called the Human - Machine Interface or simply HMI. The second microcontroller is then required to get the human wishes from HMI, “do the job” and put the results back to HMI, so that operator can see it.

Clearly, the most important form of communication between the microcontroller system and a man is the visual communication. In this chapter we will discuss various ways of displaying data, from the simplest to more elaborate ones. You'll see how to use LED diodes, Seven-Segment Displays, character- and graphic LCDs. We will also consider using BASIC for sound signalization necessary in certain applications.

Just remember: the more profound communication you wish to be, the more MCU resources it’ll take.

**7.1 LED Diode**

One of the most frequently used components in electronics is surely the LED diode (LED stands for Light Emitting Diode). Some of common LED diode features include: size, shape, color, working voltage (Diode voltage) Ud and electric current Id. LED diode can be round, rectangular or triangular in shape, although manufacturers of these components can produce any shape needed for specific purposes. Size i.e. diameter of round LED diodes ranges from 3 to 12 mm, with 3 - 5 mm sizes most commonly used. Common colors include red, yellow, green, orange, blue, etc. Working voltage is 1.7V for red, 2.1V for green and 2.3 for orange color. This voltage can be higher depending on the manufacturer. Normal current Id through diode is 10 mA, while maximal current reaches 25 mA. High current consumption can present problem to devices with battery power supply, so in that case low current LED diode (Id ~ 1-2 mA) should be used. For LED diode to emit light with maximum capacity, it is necessary to connect it properly or it might get damaged.
The positive pole is connected to anode, while ground is connected to cathode. For matter of differentiating the two, cathode is marked by mark on casing and shorter pin. Diode will emit light only if current flows from anode to cathode; in the other case there will be no current. Resistor is added serial to LED diode, limiting the maximal current through diode and protecting it from damage. Resistor value can be calculated from the equation on the picture above, where $U_r$ represents voltage on resistor. For $+5V$ power supply and $10$ mA current resistor used should have value of $330\, \Omega$.

LED diode can be connected to microcontroller in two ways. One way is to have microcontroller "turning on" LED diode with logical one and the other way is with logical zero. The first way is not so frequent (which doesn't mean it doesn't have applications) because it requires the microcontroller to be diode current source. The second way works with higher current LED diodes.
The following example toggles LEDs of PORTB every second.

**program** LED_Blinking

**main:**

TRISB = 0  
PORTB = %11111111  
Delay_ms(1000)

PORTB = %00000000

Delay_ms(1000)

goto main

**end.**

## 7.2 Seven-Segment Displays

Seven-segment digits represent more advanced form of visual communication. The name comes from the seven diodes (there is an eighth diode for a dot) arranged to form decimal digits from 0 to 9. Appearance of a seven-segment digit is given on a picture below.
As seven-segment digits have better temperature tolerance and visibility than LCD displays, they are very common in industrial applications. Their use satisfies all criteria including the financial one. They are commonly used for displaying value read from sensors, etc.

One of the ways to connect seven-segment display to the microcontroller is given in the figure below. System is connected to use seven-segment digits with common cathode. This means that segments emit light when logical one is brought to them, and that output of all segments must be a transistor connected to common cathode, as shown on the picture. If transistor is in conducting mode any segment with logical one will emit light, and if not no segment will emit light, regardless of its pin state.
Bases of transistors T1 and T2 are connected to pin0 and pin1 of PORTA. Setting those pins turns on the transistor, allowing every segment from "a" to "h", with logical one on it, to emit light. If zero is on transistor base, none of the segments will emit light, regardless of the pin state.

Using the previous scheme, we could display a sequence of nine digits like this:

```plaintext
program seven_seg_onedigit

dim i as byte

' Function mask returns mask of parameter 'num' for common cathode 7-seg. display

sub function mask(dim num as byte) as byte

    select case num
        case 0 result = $3F
        case 1 result = $06
        case 2 result = $5B
        case 3 result = $4F
        case 4 result = $66
        case 5 result = $6D
    end select

end function
```

Example of connecting seven-segment displays in multiplex mode with the microcontroller.
case 6  result = $7D
case 7  result = $07
case 8  result = $7F
case 9  result = $6F
end select
end sub

main:

INTCON = 0  ' Disable PEIE, INTE, RBIE, TOIE
TRISA  = 0
TRISB  = 0
PORTB = 0
PORTA = 2

do
  for i = 0 to 9
    PORTB = mask(i)
    Delay_ms(1000)
  next i
loop until false  ' Endless loop

end.

Purpose of the program is to display numbers 0 to 9 on the ones digit, with 1 second
delay. In order to display a number, its mask must be sent to PORTB. For example, if we
need to display "1", segments b and c must be set to 1 and the rest must be zero. If
(according to the scheme above) segments b and c are connected to the first and the
second pin of PORTB, values 0000 and 0110 should be set to PORTB. Thus, mask for
number "1" is value 0000 0110 or 06 hexadecimal. The following table contains
corresponding mask values for numbers 0-9:

<table>
<thead>
<tr>
<th>Digit</th>
<th>Seg. h</th>
<th>Seg. g</th>
<th>Seg. f</th>
<th>Seg. e</th>
<th>Seg. d</th>
<th>Seg. c</th>
<th>Seg. b</th>
<th>Seg. a</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$3F</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$06</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$5B</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$4F</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$66</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$6D</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$7D</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$07</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$7F</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$6F</td>
</tr>
</tbody>
</table>
You are not, however, limited to displaying digits. You can use 7seg Display Decoder, a built-in tool of mikroBasic, to get hex code of any other viable combination of segments you would like to display.

But what do we do when we need to display more than one digit on two or more displays? We have to put a mask on one digit quickly enough and activate its transistor, then put the second mask and activate the second transistor (of course, if one of the transistors is in conducting mode, the other should not work because both digits will display the same value). The process is known as “multiplexing”: digits are displayed in a way that human eye gets impression of simultaneous display of both digits – actually only one display emits at any given moment.

Now, let’s say we need to display number 38. First, the number should be separated into tens and ones (in this case, digits 3 and 8) and their masks sent to PORTB. The rest of the program is very similar to the last example, except for having one transition caused by displaying one digit after another:

```plaintext
program seven_seg_twodigits

dim v as byte
dim por1 as byte
dim por2 as byte

sub procedure interrupt
begin
  if v = 0 then
    PORTB = por2  ' Send mask of tens to PORTB
    PORTA = 1    ' Turn on 1st 7seg, turn off 2nd
    v = 1
  else
    PORTB = por1  ' Send mask of ones to PORTB
    PORTA = 2    ' Turn on 2nd 7seg, turn off 1st
    v = 0
  end if
  TMR0 = 0      ' Clear TMRO
  INTCON = $20  ' Clear TMROIF and set TMROIE
end sub

main:

OPTION_REG = $80      ' Pull-up resistors
TRISA = 0              ' PORTA is output
TRISB = 0              ' PORTB is output
PORTB = 0              ' Clear PORTB (make sure LEDs are off)
PORTA = 0              ' Clear PORTA (make sure both displays are off)
TMR0 = 0               ' Clear TMRO
por1 = $7F             ' Mask for '8' (check the table above)
por2 = $4F             ' Mask for '3' (check the table above)
INTCON = $A0           ' Enable T0IE

while true
  nop
wend
```
The multiplexing problem is solved for now, but your program probably doesn’t have a sole purpose of printing constant values on 7seg display. It is usually just a subroutine for displaying certain information. However, this approach to printing data on display has proven to be very convenient for more complicated programs. You can also move part of the program for refreshing the digits (handling the masks) to the interrupt routine.

The following example increases variable \( i \) from 0 to 99 and prints it on displays. After reaching 99, counter begins anew.

```plaintext
program seven_seg_counting

dim i as byte
dim j as byte
dim v as byte
dim por1 as byte
dim por2 as byte

' This function returns masks
' for common cathode 7-seg display

sub function mask(dim num as byte) as byte

    select case num
        case 0 result = $3F
        case 1 result = $06
        case 2 result = $5B
        case 3 result = $4F
        case 4 result = $66
        case 5 result = $6D
        case 6 result = $7D
        case 7 result = $07
        case 8 result = $7F
        case 9 result = $6F
    end select

end sub

sub procedure interrupt

    if v = 0 then
        PORTB = por2   ' Prepare mask for digit
        PORTA = 1      ' Turn on 1st, turn off 2nd 7seg
        v = 1
    else
        PORTB = por1   ' Prepare mask for digit
        PORTA = 2      ' Turn on 2nd, turn off 1st 7seg
        v = 0
    end if

    TMR0 = 0
    INTCON = $20
```
end sub

main:

OPTION_REG = $80
por2 = $3F
j = 0
TMR0 = 0
INTCON = $A0  ' Disable PEIE, INTE, RBIE, TOIE
TRISA = 0
TRISB = 0
PORTB = 0
PORTA = 0

do
   for i = 0 to 99  ' Count from 0 to 99
      ' Prepare ones digit
      j = i mod 10
      por1 = mask(j)

      ' Prepare tens digit
      j = (i div 10) mod 10
      por2 = mask(j)

      Delay_ms(1000)
   next i
loop until false
end.

In the course of the main program, programmer doesn’t need to worry of refreshing the display. Just call the subroutine mask every time display needs to change.

back to the top

7.3 LCD Display, 4-bit and 8-bit Interface

One of the best solutions for devices that require visualizing the data is the “smart” Liquid Crystal Display (LCD). This type of display consists of 7x5 dot segments arranged in rows. One row can consist of 8, 16, 20, or 40 segments, and LCD display can have 1, 2, or 4 rows.
LCD connects to microcontroller via 4-bit or 8-bit bus (4 or 8 lines). R/W signal is on the ground, because communication is one-way (toward LCD). Some displays have built-in backlight that can be turned on with RD1 pin via PNP transistor BC557.

Our following example prints text on LCD via 4-bit interface. Assumed pin configuration is default.

```
program LCD_default_test

dim Text as char[20]

main:

  TRISB = 0           ' PORTB is output
  LCD_Init(PORTB)    ' Initialize LCD at PORTB
  LCD_Cmd(LCD_CURSOR_OFF) ' Turn off cursor
  Text = "mikroelektronika"  ' Print text at LCD
  LCD_Out(1, 1, Text)   ' Print text at LCD

end.
```

Our second example prints text on LCD via 8-bit interface, with custom pin configuration.

```
program Lcd8_default_test

dim text as char[20]

main:

  TRISB = 0             ' PORTB is output
  TRISD = 0             ' PORTD is output
  Lcd8_Init(PORTB, PORTD) ' Initialize LCD at PORTB and PORTD
```

```
end.
```
7.4 Graphical LCD (PIC18 only)

Most commonly used Graphical LCD (GLCD) has screen resolution of 128x64 pixels. This allows creating more elaborate visual messages than usual LCD can provide, involving drawings and bitmaps.

The following figure shows GLCD HW connection by default initialization (using GLCD_LCD_Init routine); if you need different pin settings, refer to GLCD_LCD_Config.
BASIC offers a comprehensive library for GLCD – refer to Chapter 5: Built-in and Library Routines for more information. Our following example demonstrates the possibilities of GLCD and the mentioned library. Note that the library works with PIC18 only.

```basic
program GLCD_test
    ' For PIC18
include "GLCD_128x64.pbas"    ' You need to include GLCD_128x64 library
dim text as string[25]

main:
    PORTC = 0
    PORTB = 0
    PORTD = 0
    TRISC = 0
    TRISD = 0
    TRISB = 0
    GLCD_LCD_Init(PORTC, PORTD)    ' default settings
    GLCD_Set_Font(FONT_NORMAL1)
```

```plaintext
KS0108 GLCD Test
"Hello world"
mikroElektronika
```
while true
    GLCD_Clear_Screen

    ' Draw Circles
    GLCD_Clear_Screen
    text = "Circle"
    GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT)
    GLCD_Circle(63,31,10)
    Delay_Ms(4000)

    ' Draw Rectangles
    GLCD_Clear_Screen
    text = "Rectangle"
    GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT)
    GLCD_Rectangle(10, 0, 30, 35)
    Delay_Ms(4000)
    GLCD_Clear_Screen

    ' Draw Lines
    GLCD_Clear_Screen
    text = "Line"
    GLCD_Put_Text(55, 7, text, NONINVERTED_TEXT)
    GLCD_Line(0, 0, 127, 50)
    GLCD_Line(0,63, 50, 0)
    Delay_Ms(5000)

    ' Fonts Demo
    GLCD_Clear_Screen
    text = "Fonts DEMO"
    GLCD_Set_Font(FONT_TINY)
    GLCD_Put_Text(0, 4, text, NONINVERTED_TEXT)
    GLCD_Put_Text(0, 5, text, INVERTED_TEXT)
    GLCD_Set_Font(FONT_BIG)
    GLCD_Put_Text(0, 6, text, NONINVERTED_TEXT)
    GLCD_Put_Text(0, 7, text, INVERTED_TEXT)
    Delay_ms(5000)
wend
end.

back to the top

7.5 Sound Signalization

Some applications require sound signalization in addition to visual or instead of it. It is commonly used to alert or announce the termination of some long, time-consuming
process. The information presented by such means is fairly simple, but relieves the user from having to constantly look into displays and dials.

BASIC’s Sound library facilitates generating sound signals and output on specified port. We will present a simple demonstration using piezoe speaker connected to microcontroller’s port.

```plaintext
program Sound

' The following three tones are calculated for 4MHz crystal
sub procedure Tone1
    Sound_Play(200, 200) ' Period = 2ms <=> 500Hz, Duration = 200 periods
end sub

sub procedure Tone2
    Sound_Play(180, 200) ' Period = 1.8ms <=> 555Hz
end sub

sub procedure Tone3
    Sound_Play(160, 200) ' Period = 1.6ms <=> 625Hz
end sub

sub procedure Melody ' Plays the melody "Yellow house"
    Tone1
    Tone2
    Tone3
    Tone3
    Tone1
    Tone2
    Tone3
    Tone3
    Tone1
    Tone2
    Tone3
end sub

main:
```
TRISB = $F0

Sound_Init(PORTB, 2)  ' Connect speaker on pins RB2 and GND
Sound_Play(50, 100)

while true
  if Button(PORTB, 7, 1, 1) then
    Tone1
  end if
  while TestBit(PORTB, 7) = 1
    nop
  wend

  if Button(PORTB, 6, 1, 1) then
    Tone2
  end if
  while TestBit(PORTB, 6) = 1
    nop
  wend

  if Button(PORTB, 5, 1, 1) then
    Tone3
  end if
  while TestBit(PORTB, 5) = 1
    nop
  wend

  if Button(PORTB, 4, 1, 1) then
    Melody
  end if
  while TestBit(PORTB, 4) = 1
    nop
  wend
wend
end.