Programming the SX Microcontroller

A Complete Guide by Günther Daubach

2ND EDITION

WARRANTY

Parallax Inc. warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax Inc. will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax will return your product or its replacement using the same shipping method used to ship the product to Parallax.

14-DAY MONEY BACK GUARANTEE

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax Inc. will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

COPYRIGHTS AND TRADEMARKS

ISBN 1-928982-16-6

DISCLAIMER OF LIABILITY

Parallax Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax Inc. is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your BASIC Stamp application, no matter how life-threatening it may be.Internet Access

INTERNET RESOURCES

We maintain Internet systems for your use. These may be used to obtain software, communicate with members of Parallax, and communicate with other customers. Access information is shown below:

E-mail: support@parallax.com
Web: http://www.parallax.com/sx

Parallax maintains an e-mail discussion list for people interested in programming SX chips. The "SXTech" list server includes engineers, hobbyists, and enthusiasts. The list works like this: lots of people subscribe to the list, and then all questions and answers to the list are distributed to all subscribers. It's a fun, fast, and free way to discuss SX programming issues and get answers to technical questions. This list generates about 10 messages per day. Subscribe at www.yahoogroups.com under the group name "sxtech".

Table of Contents

Programming the SX Microcontroller

A Complete Guide by Günther Daubach

2ND EDITION

Section I - Tutorial

1 Section I - Tutorial

1.1 Introduction

This first part of the book is intended to give you a step-by-step introduction in how to use a development system for the SX controller, and how to write the first applications for the SX.

Development systems for the SX are offered by several vendors. In this introduction, we will describe the SX-Key development system offered by Parallax.

In this text, you will find several sections that are marked in gray, together with one of the symbols below:

> The exclamation mark indicates important information. You should read this text in any case to avoid problems.

> This symbol marks a section that contains useful additional information, which is not necessary for understanding the current topic.

> The Tutorial part of this book does not describe every feature of the SX in detail. The "R" symbol followed by a chapter and a page number indicates that more information about a topic can be found in the reference section of this book.

> This symbol marks a section that contains useful additional information, which is not necessary for understanding the current topic.

Throughout the text, we have to deal with addresses, data, and values. The SX handles and stores all these in binary format, i.e. as a collection of bits where each bit can be set (1) or cleared (0). As the data memory is organized in 8-bit registers, data is always handled in bytes, i.e. in groups of 8 bits. Instead of writing binary numbers, we will use two hexadecimal digits to represent the contents of a register in most cases. The SX program memory is addressed with 12 bit values. To represent an address, we usually use three hexadecimal digits. Sometimes, when it comes to time calculations, etc. it is easier for us human beings to do the calculations in decimal. In order to distinguish between the different number types, we use the similar notation, most of the SX Assemblers allow:

A leading "%" for binary numbers, a leading "$" for hexadecimal numbers, and no special character for decimal numbers, e.g.

%1011 1100 = $BC = 188

Sometimes, you may also find a notation like 0xbc, an alternative notation for hexadecimal numbers. C programmers are used to it quite well. Most available Assemblers for the SX also accept the “postfix” notation for binary, and hexadecimal values, e.g. 10010101B, or FFH, but we will not use this format in the book text.

In the tutorial example programs, we sometimes make use of instructions that are not always explained when they are used first. Please refer to the "Alphabetic Instruction Overview" in the Quick Reference section of this book when you want to learn more about the function of a specific instruction.

1.2 SX Development - What You Need

1.2.1 The Tools

When you plan to develop software and hardware for a new type of microcontroller, you usually need to buy new "tools", meaning a financial investment. For the SX, this is the case as well but fortunately, several vendors offer moderately priced development systems for the SX.

One reason why development systems for the SX can be offered cheaper than systems for other microcontrollers lies in the SX itself: It has "built-in" debugging capabilities, and due to the EEPROM program memory, and the in-system programming features, there is no need for UV EPROM erasers or in-circuit emulation systems.

1.2.2 Prototyping Systems

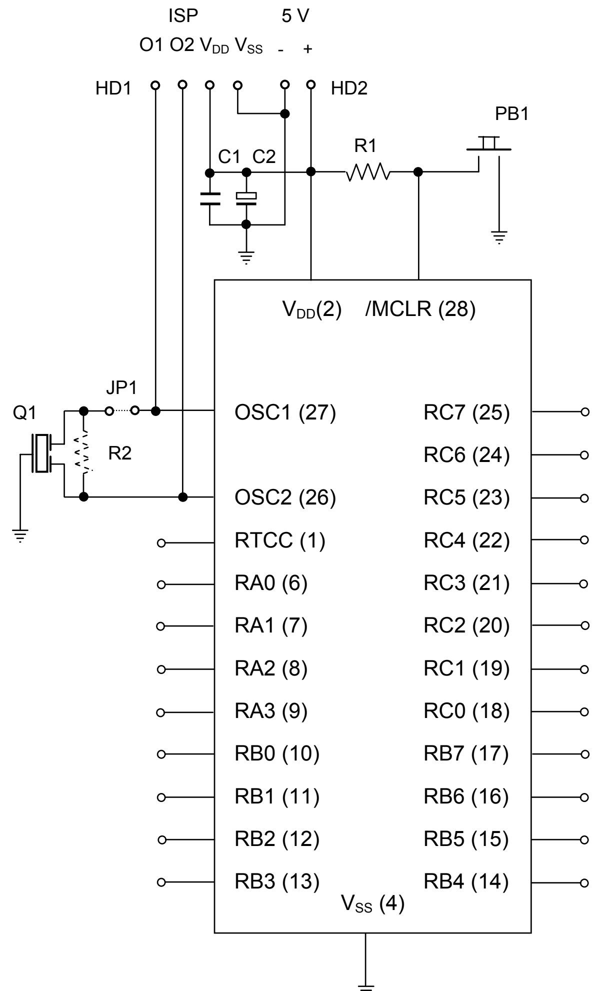
When you perform your first experiments with the SX, it is most likely that you do not have a finished PCB on hand, designed for the system you intend to develop. Various prototype boards are offered by Parallax, Ubicom, and other vendors.

All the boards come with the basic components that are required to get the SX up and running, like a voltage regulator, a reset circuitry, a clock generator, etc. Additional components like LEDs, switches or pushbuttons, RS-232 drivers, serial EEPROMs, components for A/D conversion, and filters for PWM outputs can be found on most of the boards as well. Ubicom also offers boards designed to test a specific SX feature in detail, like communications via the I²C bus, or a demonstration board for TCP/IP applications (this is a WEB server on a 4.5 by 8.5 mm PCB).

1.2.3 A "Home-brew" Prototyping System

!	Like all CMOS components, the SX can be damaged by excessive voltages produced by electro-static discharges. Therefore, take the usual safety measures that are required when handling static sensitive components. Also, make sure that the supply voltage does not exceed the maximum value specified in the SX datasheet (7.5 Volts).

For the first experiments, you can build your own “homebrew” prototyping system as shown in the schematic below:



A simple SX Prototyping System

Bill of Material

Name	Dimension	Remark
R1	10 kΩ	
R2	Depends on the resonator, or crystal type	
C1	100 nF	Filter capacitors, use two or more if necessary.
C2	100 µF Tantalum	
PB1	Pushbutton	Reset
Q1	50 MHz Ceramic resonator	Alternatively you may use a 50 MHz crystal
IC1	SX 28, DIP package	On a PCB, use a socket
JP1	Jumper	Open when the development system is connected to HD1
HD1	4-pin header connector, 1/10'' spacing	Connector for development system
HD2	2-pin header connector	5V stabilized

In order to connect external components to the SX, you should provide header pins for the port lines, and for the RTCC input.

Take care that the leads connecting to OSC1 and OSC2 are as short as possible as the clock frequency may be up to 75 MHz (depending on the SX type used). It is also important to filter V_{DD} by placing capacitors as close as possible to the V_{DD} and V_{SS} pins of the SX.

1.3 SX-Key Quick Start using the Parallax SX-Key

1.3.1 The SX-Key

Parallax, Inc. has developed SX-Key® development system for the SX. The major component is a small PCB with a female 9-pin SUB-D connector to be connected to a serial COM port of a PC at one end, and with a 4-pole plug at the other end that connects to the 4-pin ISP header you will find on all prototype boards.

> If you have built your own prototype board, double check that the header pins on your board are connected to the SX pins in an order that matches the one printed on the SX-Key plug, i.e. OSC1-OSC2-V_{DD}-V_{SS}. As this plug is not indexed, make sure that it is plugged in the right direction.

When you take a closer look at this small PCB, you will notice that it is crowded with various components, including an SX controller, a clock generator and a voltage converter that provides the programming voltage for the SX under test.

Together with the PC software which comes with the SX-Key you have a complete development system allowing you to write applications for the SX in Assembly language, program the SX, and test the application using the integrated debugger, or running the application “stand-alone”. All this is performed using the target SX on the prototype board, and not in a somehow simulated mode. This means that you can run an application in real-time speed but also in single steps for testing purposes. All this is controlled by the PC program via a serial connection.

1.3.2 Installing the SX-Key IDE Software

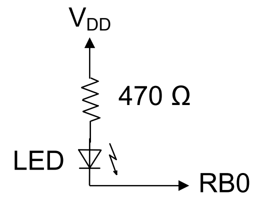
Together with the SX-Keyprogramming tool, you should have received a CD-ROM with the SX-Key IDE software. You need a PC running a Windows-OS (Win 95 or greater). To install the software, run the setup program that is on the CD-ROM.

It makes sense to setup a shortcut to launch the SX-Key software from the desktop or from the Start menu.

Before taking the next steps, it is a good idea to visit Parallax's WEB Site (www.parallax.com) looking for newer versions of the SX-Key software. Parallax, Inc. usually offers new versions for download free of charge.

1.3.3 The First Program

For the first tests, we need an LED connected to port pin RB0 of the SX:



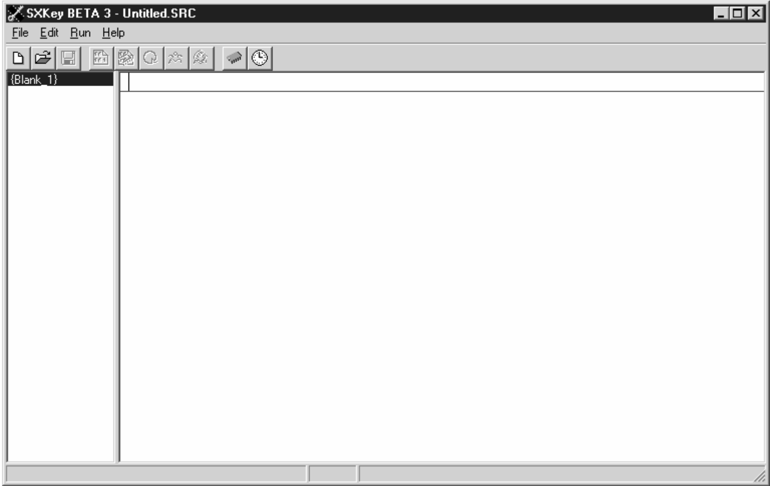
Some commercially available prototype boards do have an LED installed like this at the RB0 pin already. On some boards, the LED may be connected between RB0 and V_{SS} instead. For our first tests, this does not matter.

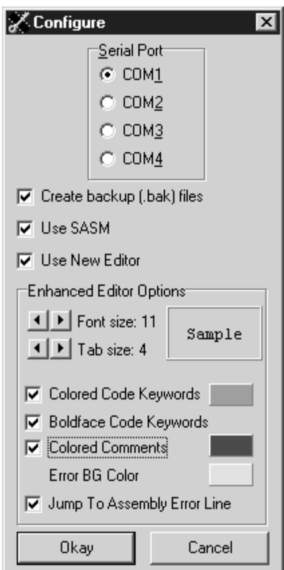
When you use a Parallax SX Tech board, you can easily position the two required components in the breadboarding area.

If not yet done, connect the SX-Key via a cable to a serial port of your PC and plug the other end on to the ISP header pins (double-check the correct orientation of the plug). Make sure that you use a "straight-through" type of serial cable, i.e. not a null-modem cable. If you need adapters to convert between 9-pin and 25-pin DB connectors, also make sure that the adapters are "straight-through". Finally, note whether the cable is connected to COM1, COM2, or any other COM port.

Make sure that the jumper that connects a resonator or crystal to the OSC1 pin on the prototype board is open. If there is no jumper, remove the resonator or crystal from the socket.

Now connect the power supply to the prototype board, and launch the SX-Key software on the PC. After the program has loaded you should see a window like this:





This window shows the text editor of the new Parallax SX-Key IDE, Version 2. Compared to former versions, this new IDE has a lot of useful enhancements, like syntax highlighting, the possibility to open several files at the same time, and much more. You will use the editor to enter the application source code.

Select "Configure" from the "Run" menu, or press Ctrl-U as a shortcut to open the dialog box shown below:

Click a radio button in the “Serial Port” section to select the COM port you want to use for the SX-Key. For now, leave the other options in this dialog unchanged. See the Parallax documentation for the meaning of the other options. Finally, click "Okay" to close the dialog box.

Next, enter the following text in the editor window:

```
; =================================================================
; Programming the SX Microcontroller
; TUT001.SRC
; =================================================================
LIST Q = 37
DEVICE  SX28L, TURBO, STACKX, OSCHS2
IRC_CAL IRC_FAST
FREQ    50_000_000
RESET   0

  mov    !rb, #%11111110
Loop
  clrb   rb.0
  setb   rb.0
  jmp    Loop
```

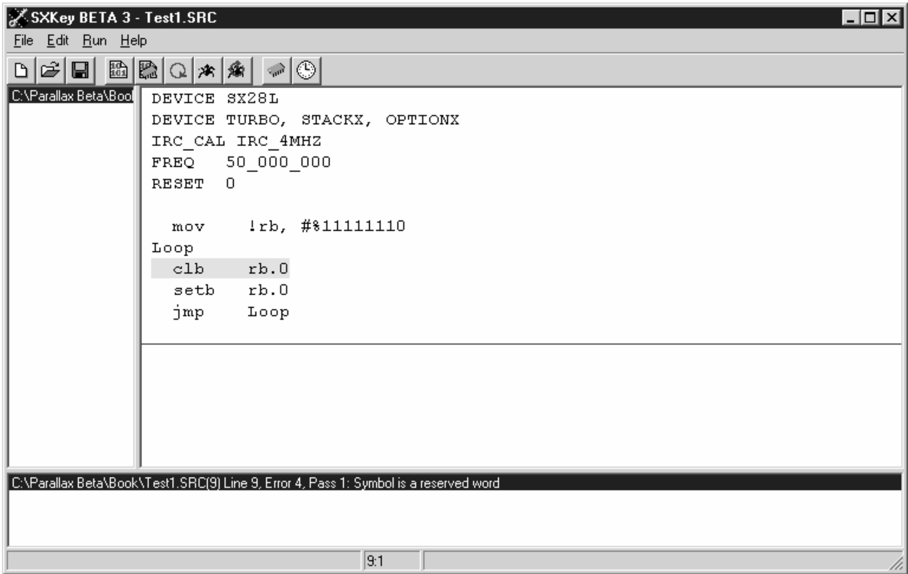
It makes no difference if you type uppercase or lowercase letters. You may enter tabs or spaces to separate the words. The leading space we have inserted in some lines is not really required but it makes the text look a bit more "structured". You may also insert any number of empty lines at any place in order to structure the source code text.

> This and the following sample programs assume that you are using an SX 28 controller. In case you are using another SX type, replace the **`DEVICE SX28L`** directive with the respective **`DEVICE`** directive.

After you have entered your first program code, you need to save it. Either click on the diskette shortcut button, select “Save” from the “File” menu, or enter Ctrl-S on the keyboard to open the “Save Source as...” dialog. If you like, select or create a folder where the file shall be saved, and then enter the file name, e.g. TUT001 (the “.SRC“ extension will be added automatically, so there is no need to type it in).

Next, click the Assemble button (the fourth button from the left), select “Assemble” from the “Run” menu or press Ctrl-A as a shortcut to assemble this little program. When a dialog opens, telling you that the file needs to be saved prior to assembling, click the “Ok” button. You may consider to mark the “Don’t show this dialog again” option to avoid that it pops up whenever you assemble another program. When the status line at the bottom right of the editor window displays "Assembly Successful", the program could be assembled without errors, and it is most likely that you can execute it.

Should a dialog box pop up containing the message “Unable to assemble due to errors in source code”, click the “Ok” button. The editor window should then look similar to the next figure.



It is most likely that the error is caused by some misspelled text. The offending line is highlighted, and in the new window that has opened below the text, you will find a description of the error. In our example, the assembler error message reads, "Symbol is a reserved word". This may be a bit miss-leading, but assemblers don’t have too much intelligence to detect each and every reason for an error. In our example, the word "clb" left of "rb" is misspelled, because it should read "clrb".

Make the necessary corrections, and press Ctrl-A again to assemble the modified program until the message "Assembly Successful" finally shows up in the status line.

After having corrected any errors, you should again save your "masterpiece". Click the “Save” button (the button showing the diskette symbol), select "Save" from the "File" menu, or press the shortcut Ctrl-S.

When source code files become larger, it is a good idea to save them from time to time. Simply click the Save button or press the Ctrl-S shortcut to make sure that your work is not lost. Please note that the editor automatically generates a backup copy of the previously saved version of a saved file when the option “Create backup (.bak) files” is activated in the Configure dialog. This means that you will always have the current and the previous version of a program available on your disk drive. In order to archive certain versions of a source code file, use the "File - Save As" option to save the file under another name.

Now click the Debug button (the fourth button from the left with the bug symbol), select "Debug" from the "Run" menu, or type the Ctrl-D shortcut to launch the SX-Key debugger. As programs are always executed on the target SX controller, they must be transferred to the SX before debugging can begin. Invoking the debugger means that the current version of the source code file is assembled, and then the resulting machine program is transferred to the SX (provided that there are no errors in the source code).

When you see a display like

you are very close to executing your first program.

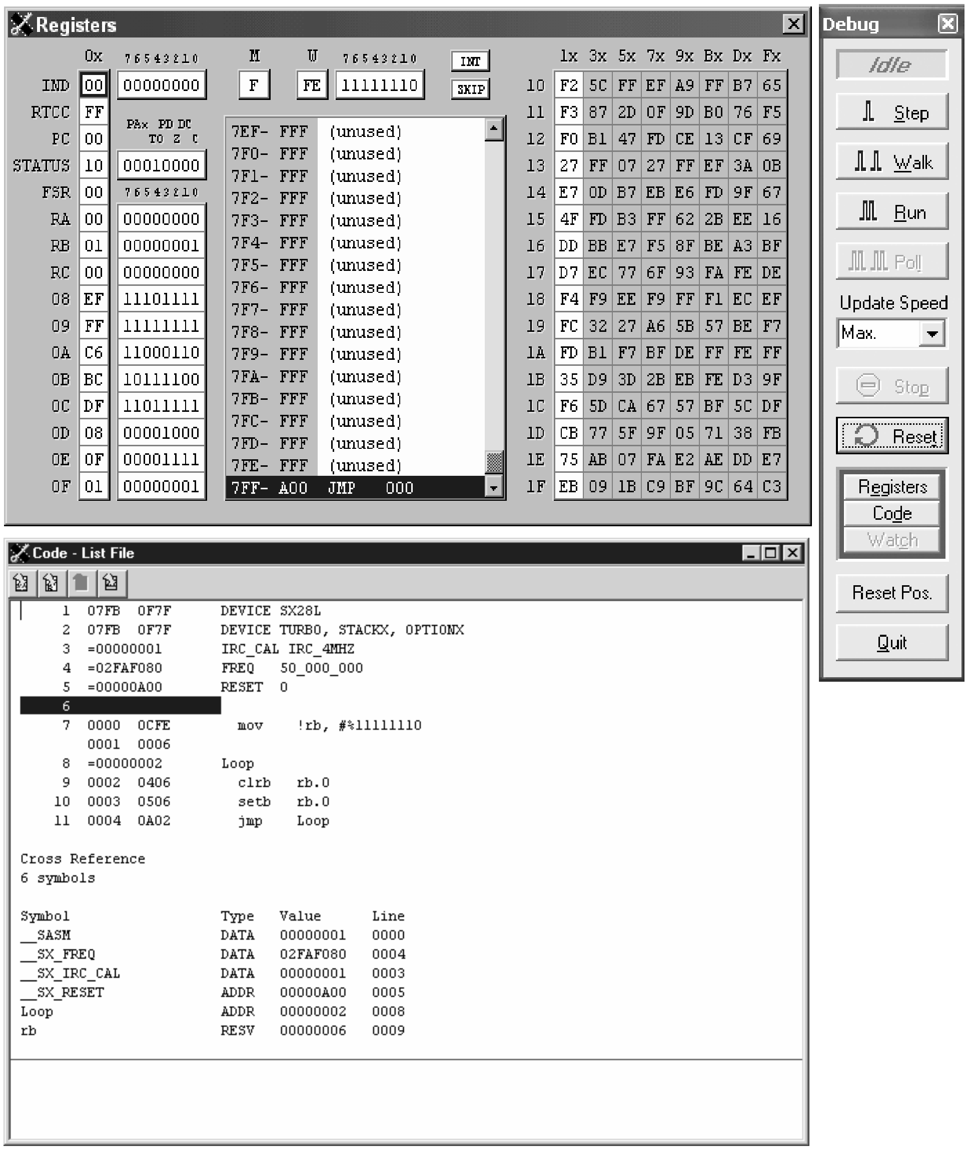
Should an error message show up like this one:

click the "Abort" button, and find out what the problem is. There are several reasons for such message:

- No supply voltage connected to the prototype board, supply voltage too low, or too high.
- SX-Key not connected to the specified COM port, or wrong orientation of the ISP plug.
- The jumper between SX-Key pin OSC1 and the resonator is still in position.
- The orientation of the SX in the socket is wrong, is not plugged in at all, or a pin has been bent.
- The SX is defective.
- The SX-Key is defective.

Check the reasons in the given order. Hopefully, you will have found the reason before reaching the last two items in the list.

After you have corrected the problem, press Ctrl-D again. Within a few seconds, the "Programming" message should come up. When the program has been transferred to the SX, the message box is closed, and the following windows will open:

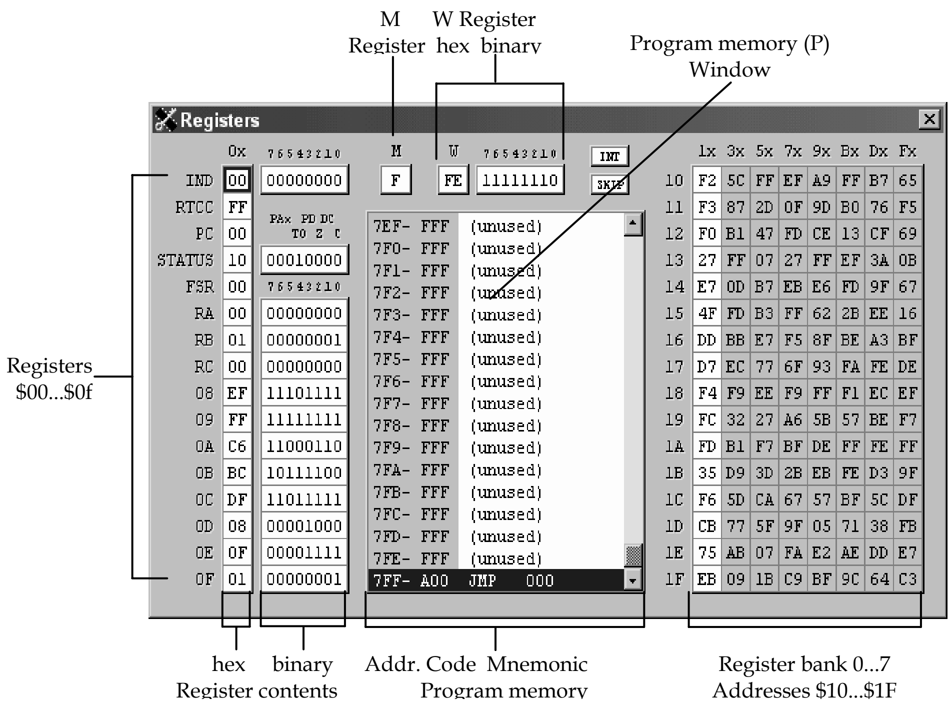


The size and the position of these windows depends on the screen resolution you have configured. You may move each of the windows, and you can also resize the "Code - List File" window if you like.

1.3.4 The SX-Key Debugger Windows

1.3.4.1 The Registers (R) Window

This window shows the SX "internals", i.e. its various registers. The figure below explains the different areas in that window:



In the middle of the "Registers" window, there is another area, that we will call "Program Memory" or "P" Window. Currently address $7FF is highlighted in this window. (Note that the R window displays all values in hex or binary without leading "$" or "%" signs.)

1.3.4.2 The Code – List File (C) Window

This window displays the assembly source code as you have entered it, plus some additional information. Actually, this is the so-called "List" file format showing the machine codes that were generated by the Assembler, together with the addresses where the machine codes are stored in program memory.

1.3.4.3 The Debug (D) Control Window

This window contains the buttons that are required to operate the debugger, and other buttons to open the debugger windows in case they have been closed or minimized. Click one of the buttons "Registers", "Code", or "Watch" to open the corresponding windows ("Watch" is inactive for now).

In the following text, we will use the short form "D", "R" or "C" to refer to one of the windows, and "P" for the program memory display in the center of the "R" window.

1.3.5 Executing the First Program in Single Steps

The highlight in the P window is positioned at program address $7FF and in the C window the line containing **`RESET`** 0 is highlighted. After a reset, the SX controller sets the program counter, i.e. the register that contains the address of the instruction to be executed next to the address of the highest available program memory location. For the SX 28, this is $7FF.

At this address, the assembler has inserted an instruction that makes the SX continue program execution at (i.e. jump to) the address defined by the **`RESET`** directive in our program (0 in our example).

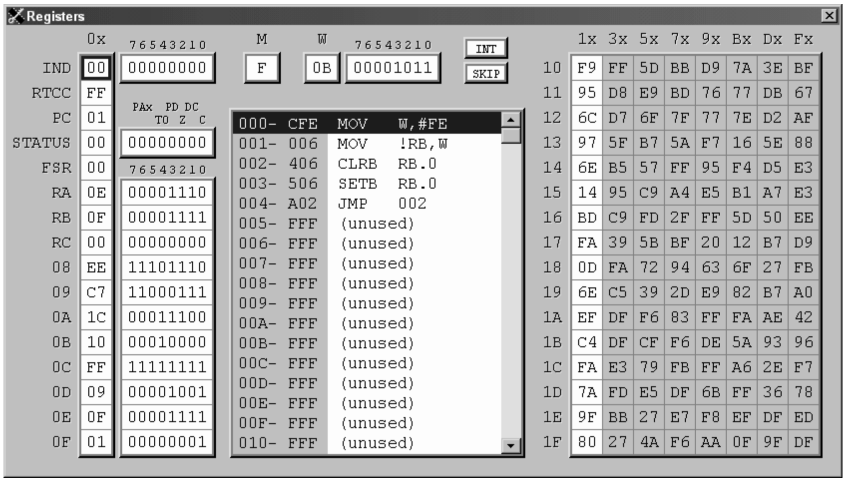
The instruction which unconditionally causes the program execution to branch to another location is the **`jmp`** instruction. This instruction loads the program counter (PC) with the target address, i.e. the PC then "points" to the first program instruction to be executed.

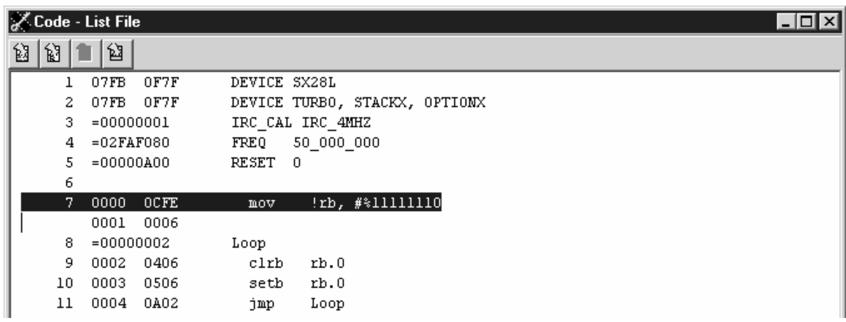
By the way, you can use the scroll bar to the right of the P window to scroll the displayed text up and down. In this case, the highlighted line may be moved out of the window, but it keeps its position to mark the next instruction to be executed.

The same is true for the C window. Here you have horizontal and vertical scroll bars available to move the text.

Now left-click the "Step" button in the D window once. Alternatively, you may also enter Alt-S on the keyboard (make sure that the D window has focus in this case).

Now the window contents should change like this:





The highlight has moved to address $000, i.e. the SX has executed the **`jmp 0`** instruction that is located at address $7ff which caused the jump to the new program address at $000. As you can see, the highlight in the (C) window is now positioned on that line.

1.3.6 Compound Instructions

The P window displays the **`MOV W, #FE`** instruction where the C window has highlighted the **`mov !rb, #%11111110`** instruction, i.e. a different instruction.

The point here is that the SX does not "know" how to execute a **`MOV W, #FE`** instruction. The Assembler automatically has generated two separate instructions that are “familiar” to the SX:

```
MOV W, #FE
MOV !RB, W
```

These two instructions were saved in two subsequent locations of the program memory. We will call such assembler instructions *compound instructions*. There are various compound instructions available to make programmer's life a bit easier because it saves you the extra work of writing two or more separate lines of code. (Later, we will see that compound statements can also cause situations to make programmer's lives quite hard.)

Now let's find out what the **`mov !rb, #%11111110`** instruction means. A **`mov`** instruction copies the contents of one register into another register or it copies a constant value into a register. "mov" is derived from "move", but it actually copies a value instead of moving it, i.e. the contents of the source remains unchanged after execution, where the target receives the new value.

The hash mark "`#`" left of the binary number %11111110 means that the constant value %11111110 shall be copied into a target called !rb here. The hash-sign is very important - if you leave it out, the assembler will assume that you want to copy the contents of register %11111110 to !rb instead!

"!rb" specifies the SX configuration register for port B. We will discuss port configuration in more detail later. For now, you should keep in mind that a cleared bit in the configuration register means that the associated port pin will become an output. To make clear which bits are set and cleared in the !rb register, we use binary notation here.

Here, we configure the RB.0 pin as an output - this is where we have connected the LED.

Actually, the **`mov !rb, #%11111110`** is composed by the two instructions

```
mov w, #%11111110
mov !rb, w
```

i.e. the constant value %11111110 is copied into the w register, and then the contents of w is copied into !rb. The w register (the "Working" register) is used as temporary storage by many compound instructions. In general, w is a multi-purpose register used to hold one operand of arithmetic or logical instructions, to hold the result of special **`mov`** instructions with arithmetic/logical functions and as temporary storage like in the example above. The w register is similar to the "accumulator" found in other microcontrollers or microprocessors.

Now left-click the "Step" button once again, and you will notice that the highlight in the P window has moved to the second part of the compound instruction where the highlight in the C window did not move at all.

Click "Step" again to execute the **`mov !rb, w`** instruction, and to bring the highlight on to the **`clrb rb.0`** instruction. Click "Step" to let the SX execute this instruction as well, and check if the LED turns on.

If you have a prototype board where the LED is connected to V_{SS} with the other end (the cathode), click "Step" once more to execute the **`setb rb.0`** instruction that should finally turn on the LED in this case.

If you managed to turn the LED on, you are all set - your first SX program works as expected!

In case the LED remains off, check the following:

- Is the constant that is moved into !rb actually %11111110 (did you eventually forget the leading "#" or "%" characters)?
- Do all instructions address the right port (rb)?
- Do the **`clrb rb.0`** and **`setb rb.0`** instructions both contain the "0", and not another digit?
- Is the LED really connected to pin 10 (RB0) of the SX 28?
- Is the polarity of the LED correct?

In case you have found an error in the program code, click the "Quit" button to return to the editor, make the necessary corrections, and enter Ctrl-D to launch the debugger again (the program will be assembled and transferred to the SX automatically).

If the problem was caused by hardware, you have (hopefully) disconnected power from the SX. This has caused the SX-Key software to display the error message "SX-Key not found on COMx". Click the "Abort" button to return to the Editor window and re-connect the power supply then.

If the program did work properly, instead of disconnecting the power supply, click the "Quit" button to leave the debugger back to the Edit window.

When you want to re-activate the debugger again later, it is not necessary to transfer the program to the SX again as long as you did not make changes in the source code. In this case, don't select "Run - Debug", and don't press the Ctrl-D shortcut. Instead, select "Run - Debug (reenter)", or press the shortcut Ctrl-Alt-D. This starts up the debugger immediately without transferring the program into the SX.

When the debugger is active again, address $7ff is highlighted as it was when you started the debugger the first time, and you may continue the debugging session.

Now let's find out what makes the LED turn on or off:

The **`clrb rb.0`** instruction clears a bit in the specified register (rb in this case), where rb is a predefined name the assembler "knows". The assembler replaces it with $06, the address of the Port B data register. Instead of "rb", you could have written "$06" as well. Which bit shall be cleared is specified by the digit that follows the register name, separated by a period.

In our case, we clear bit 0 in the Port B data register. As the associated port pin has been configured as an output, this means that the output pin goes to low level and the LED is turned on.

The next instruction `setb rb.0` does just the opposite of `clrb` - it sets a bit in the specified register. In our case, it causes the output pin RB0 go to high level that turns the LED off again.

In the left part of the R Window, the contents of the first 16 registers are displayed in hexadecimal and binary format. When you watch the contents of address $06 you will notice how the content changes as you keep clicking the "Step" button. Note that the hexadecimal value is displayed with a red background when it has recently changed, and that bit 0 in the binary display area is also shown with a red background when its state has changed. This helps you to quickly find out which data has changed after executing an instruction.

Notice that the contents of PC is always displayed with a red background as the program counter always changes its contents, either to address the next instruction in sequence, or another location after a `jmp`, `skip` or `call` instruction.

1.3.7 Symbolic Addresses – Labels

The last instruction in our program is a `jmp` instruction that sets PC back to address $002 where the `clrb` instruction is stored. If you look at the P window, you notice that it shows `jmp 002`, but in our program, we have written `jmp Loop`.

We could also have written `jmp $002` instead, but this would not be very flexible. Imagine what happens if we would insert another instruction immediately following the `mov !rb, #%11111110` instruction. This would "shift" all subsequent instructions "up" in program memory, and to reach the `clrb rb.0` instruction, you would have to change the $002 address parameter of the `jmp` instruction. Think what a "nice job" it would be to correct all the `jmp` instructions in a program consisting of hundreds of lines, and what could happen if you would forget to correct even one of them.

Fortunately, the assembler allows the definition of symbolic addresses that makes life a lot easier. When the assembler finds a word at the beginning of a line that is neither an instruction nor another "reserved word" (more about this later), it interprets it as a "label". In this case, it stores the word (`Loop` in our example) in an internal table (the symbol table) together with the address of the instruction that follows the label, either in the same line, or in the next line that is not empty, and does not contain a comment only (`$002` here).

Whenever the assembler finds an instruction that is not followed by a numeric value, as in `jmp Loop`, it searches the symbol table for that word and replaces it with the numeric value that is stored there (`$002` in our example).

> SASM, the default assembler of the SX-Key 2.0 software expects that labels always begin in the leftmost column of a line, where other assemblers like the “old” SX-Key assembler accept leading white space. For compatibility reasons, it is a good idea to always let labels begin in the first column.

1.3.8 Running the Program at Full Speed

If you are tired of clicking the "Step" button, you might consider letting the program run at full speed. Click the "Run" button, and see what happens: The LED "glows" rather dim and not at about half of the full intensity as you might have expected.

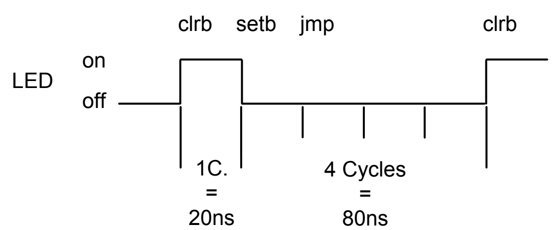
There are two reasons for this phenomenon: Duty cycle and speed.

Here are the instructions that are continuously executed while the SX runs at full speed, together with the required clock cycles:

```
Loop
  clrb   rb.0       ; 1
  setb   rb.0       ; 1
  jmp    Loop       ; 3
```

> You may add comments (like the number of clock cycles in the lines above) using the semicolon. The assembler will ignore all text in a line that follows a semicolon. If a line begins with a semicolon, all the rest of the line will be ignored. This is sometimes helpful to temporarily "comment out" instructions in a program.

The **clrb** and **setb** instructions take one clock cycle each, and the **jmp** requires three cycles. The diagram below shows the LED timing (assuming that you run the SX at 50 MHz clock):



After the **clrb** instruction, the LED is turned on. It takes one clock cycle (20 ns at 50 MHz system clock) until the **setb** instruction is executed, i.e. until the LED is turned off again. Then it takes three cycles to execute the **jmp** and another cycle for **clrb** until the LED is turned on again. This means that during 20% of one loop the LED is on, and 80% of the loop, the LED is off.

To extend the LED's on time, we need to add some "cycle eater" instructions between the **clrb** and **setb** instructions that "steal" three clock cycles. The SX "knows" a special "do nothing" instruction that exactly does this, the **nop** (for no operation).

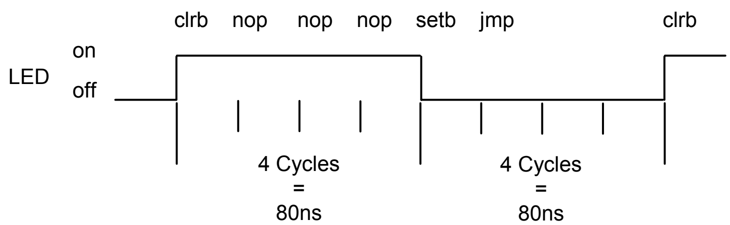
Quit the debugger, and add three **nop** instructions like this:

```
Loop
```

```
clrb   rb.0      ; 1
nop              ; 1
nop              ; 1
nop              ; 1
setb   rb.0      ; 1
jmp    Loop      ; 3
```

As you did change the source code, you can't restart the debugger now, therefore press Ctrl-D to re-assemble the modified program, and to have it transferred into the SX. Then start the program at full speed by clicking the "Run" button.

The changes result in the following timing:



Now the LED is on and off for an equal time, i.e. it has a duty cycle of 50%.

One full loop now has a period of 8 clock cycles or 160 ns, and this means a frequency of 6.25 MHz! This is a frequency LEDs are not designed for, and this is the second reason why the LED may be darker than expected.

Instead of having the LED "blink" at this frequency, we want to make it blink slowly enough so that we really can see it blink. Before we enhance the program, try the following:

First, stop the full-speed execution by either clicking the "Stop" or "Reset" button. Now click the "Walk" button, and you will see the LED blink.

The "Walk" mode is similar to single stepping except that the debugger "clicks" the "Step" button for you a couple of times per second. As you may notice, the LED does not really blink, instead it "flickers". This is because the debugger needs some time to update the window contents between each step.

Click "Reset" or "Stop" to end the "Walk" mode. When you click "Reset", the SX is really reset, i.e. the PC is reset to $7ff (and some other registers are initialized to specific values as well). When you click "Stop", the execution stops at the instruction which was executed when you clicked that button, and all registers reflect the status at that time, and you may continue program execution from that point.

1.3.9 Program Loops for Time Delays

Now we will slow down the SX in order to obtain a nicely blinking LED while the program is running at full speed.

In the last version of our program, we used three nop instructions to "eat up" time. In order to "waste" more time we will add some more program loops.

Don't enter the following statements, as we only need them for some time calculations:

```
Loop
  decsz  $08              ; 1/2
    jmp Loop              ; 3
  clrb   rb.0             ; 1
Loop1
  decsz  $08              ; 1/2
    jmp Loop1             ; 3
  setb   rb.0             ; 1
  jmp    Loop             ; 3
```

In this code, we have added two more program loops, one following the **clrb** instruction, and one following the **setb**.

Within the loops, we use the **decsz** (decrement and skip on zero) instructions. This instruction decrements the contents of the specified register (at address $08 in data memory here) by one. In case the content of the register ends up in zero after the decrement, the next instruction will be skipped (the **jmp** in our example).

Let's assume that the register at $08 contains zero when we start the program. In this case, the first **decsz** instruction would change its contents to $ff or 255. Because its content is not zero, the next instruction will not be skipped, i.e. the **jmp** instruction will be executed.

> You may wonder why 0 - 1 results in 255 in the SX, and not in -1, as you might guess. This is because the SX (like most other controllers) does not “know” about negative numbers.
>
> To understand the “underflow” from 0 to 255, let’s see what happens when a value of 255 (or %11111111) is incremented by one. The “real” result would be %100000000, i.e. the 9th bit would be set, and all other bits cleared. As the registers in the SX can only hold eight bits, the exceeding 9th bit is lost. This means that 255 + 1 yields in 0 in the SX.
>
> Decrementing a value of 0 by one is the reverse of incrementing a value of 255 by one, and this is why 0 - 1 yields in 255 in the SX.

This sequence is repeated until the content of $08 finally reaches zero. Now the **jmp** will be skipped, and the **clrb** or **setb** instructions are executed.

Again, we have added the number of clock cycles that each instruction requires. Note that the **`decsz`** instructions usually take one cycle (when there is no skip), but two in case of a skip.

As a data register can hold 256 different values (0...255), each of these loops is executed 256 times where 255 times, the skip is not performed. Therefore, each loop takes 255 * (1+3) + 2 = 1,022 clock cycles and one more cycle to clear or set the port bit. By adding the three more cycles for the final **`jmp Loop`** instruction, we end up in 1,023 * 2 + 3 = 2,047 cycles that take 2,047 * 20 ns = 40.94 µs which results in an LED blink frequency of 24.426 kHz - far beyond visibility!

Even if we would nest another delay loop within each of the two loops, the SX would still be too fast. Before considering to reduce the system clock rate, try this program:

```
; ==================================================================
; Programming the SX Microcontroller
; TUT002.SRC
; ==================================================================
include "Setup28.inc"

RESET  0
  mov   !rb, #%11111110
Loop
  decsz $08             ; 1/2
    jmp Loop            ; 3
  decsz $09             ; 1/2
    jmp Loop            ; 3
  decsz $0a             ; 1/2
    jmp Loop            ; 3
  clrb  rb.0            ; 1
Loop1
  decsz $08             ; 1/2
    jmp Loop1           ; 3
  decsz $09             ; 1/2
    jmp Loop1           ; 3
  decsz $0a             ; 1/2
    jmp Loop1           ; 3
  setb  rb.0            ; 1
  jmp   Loop            ; 3
```

Don't try to assemble, or run the program yet. Note the **`include "Setup28.inc"`** directive at the beginning of the code. The new SX-Key 2.0 software now allows to include one or more files within the code. This means that the assembler will open the file that is specified with the **`include`** directive, read its contents, and inserts the lines read at the location of the include directive. This feature is handy to add lines that are the same for many different programs.

Please create a new file in the SX-Key Editor environment, enter the following lines, and save it under the name "Setup28.inc" in the same directory where you have saved TUT002.SRC:

```
; ================================================================
; Programming the SX Microcontroller
; Setup28.inc
; ================================================================

LIST Q = 37
DEVICE  SX28L, TURBO, STACKX, OSCHS2
IRC_CAL IRC_FAST
FREQ   50_000_000
```

As we need to make some definitions concerning the configuration of the SX chip in each and every program, these configuration lines are good candidates for an include file (in the next chapter, chip configuration is explained in more detail).

We will use Setup28.inc together with many other code samples in this tutorial section of the book. Should you use another SX chip, e.g. an SX20, you only need to change the DEVICE specification in Setup28.inc, and all programs using this include file will be automatically configured for an SX20 (when you assemble them with this modified include file).

After having saved the file, assemble the second tutorial program TUT002 that you have entered before.

In this program, we have nested three program loops before executing the **`clrb`** or **`setb`** instructions and we use three registers as delay counters ($08, $09, and $0a).

In each loop, we first decrement $08 until it is zero and then decrement $09. If the content of $09 has not yet reached zero, we repeat the 256 loops decrementing $08 until $09 is zero. Then we decrement $0a, and repeat the previous steps until finally $0a is zero.

Now let's figure the approximate time the three nested loops take:

As the "inner" loop is identical to the one in the previous code example, it takes 1,022 cycles to execute it. The "middle" and the "outer" loop are executed 256 times as well, therefore the total number of clock cycles required by the three nested loops is approximately $256 * 256 * 1.022 = 67 * 10^6$ cycles. For a complete LED on-off cycle, the total time delay is $2 * 67 * 10^6 * 20\text{ns} \approx 2.6$ seconds, i.e. the resulting LED blink frequency is about 0.38 Hz.

After you have entered this new version in the editor window, don't forget to save it under a new file name (e.g. TUT002.src), and then press Ctrl-D to launch the debugger.

Click the "Run" button to execute the program. Provided that you have correctly entered the program, the LED should now blink quite slowly.

While the program is running at full speed, the R, P, and C windows are not updated because this would slow down execution far beyond real-time. If you want to take a "snapshot" of the current register status, click the "Poll" button at any time while the program is executed at full speed.

When you want to execute this program in single steps, keep in mind that one nested delay loop now takes about 67 Million steps. Maybe that clicking the "Step" button 67 Million times is a good test for your left mouse button, but we don't take any responsibility for your hurting fingers.

Even the "Walk" mode takes far too long to execute one LED on-off cycle.

As such kind of loops can be found frequently in SX programs, there should be a way to "skip over" such loops and start singe-stepping from there. Fortunately, the SX debugger allows setting a "breakpoint" that solves this problem.

1.3.10 Setting a Breakpoint

Setting a breakpoint means that you “tell” the debugger to execute the instructions beginning at the address, PC is currently pointing to, up to and including the instruction where you have set the breakpoint.

To set a breakpoint, first make sure that the program is halted by either clicking the "Stop" or "Reset" buttons. Then, in the C window, move the mouse pointer to the program line where you want to set the breakpoint and hit the left mouse button once. The debugger will display this line with a red background now, indicating that a breakpoint is active on that line. If necessary, scroll the text in the window up or down until the line you want is visible before setting the breakpoint.

In case the line with the breakpoint is the next line to be executed as well, only the left part of the line is marked with a red background while the rest of the line is highlighted with a blue background.

For example, click "Reset" for a "clean start", and then click on the line with the **`clrb rb.0`** instruction. Finally, start the program at full speed with "Run".

You will notice that it takes a while until the LED is turned on. Once the LED is on, program execution halts due to the active breakpoint, and the **`decsz $08`** instruction in **`Loop1`** is the next one to be executed.

If you like, you may single-step the program for a while but you can also click "Run" again to go through the program at full speed until the breakpoint is reached the next time. During that time, you will notice that the LED is turned off after a while, and finally is turned on again, when the program "hits" the breakpoint after executing the **`clrb`** instruction.

Please note that there can only be one breakpoint active at a time. This is not a limitation of the SX-Key software but by the SX itself. As soon as you click another line in the C window, this new line will be highlighted in red, and the line marked before is reset to standard.

In order to remove an active breakpoint, simply click on the highlighted line once again.

Please note that due to the internal structure of the SX the instruction in the line marked for a breakpoint will be executed before the program actually halts. This may be confusing sometimes, when you set a breakpoint on a line with a jump or call instruction as you will see later. In addition, the SX does not stop program execution when the breakpoint is set to a line containing a `nop` instruction.

You may add a `BREAK` directive to the source code immediately before the instruction where the debugger shall activate a "default" breakpoint when it is invoked. You can change the position of the `BREAK` directive in the source code later without the need to re-assemble the code, i.e. you may re-start the debugger using the Debug (reenter) option, or by entering the Ctrl-Alt-D shortcut.

When the debugger is active, you may change the position of the breakpoint at any time by clicking on another line in the list window that contains executable code.

1.3.11 Where to Go Next

This ends the Quick-Start chapter for the SX key development system. In this chapter, you have learned some basic SX instructions, and programming techniques but most of the chapter was dedicated to the SX-Key development system.

In the next tutorial chapters, we will concentrate on more SX instructions and features, assuming that you are familiar with the development tool, you are using: The SX-Key system.

> When you are done with debugging a program, you may want to have the SX execute the program "stand-alone", i.e. without the SX-Key probe connected. In order to do so, it is important that you re-program the SX without the debug code. Select "Program" from the "Run" menu, click the "Program" shortcut button, or enter Ctrl-P. This instructs the SX-Key to transfer the "stand-alone" code into the SX program memory. It is also necessary that the SX has another clock source now. You can find more information about the various clock sources in chapter 1.14.