PROGRAMMING The SX Microcontroller

by Günther Daubach

Warranty

Parallax warrants its products and printed documentation against defects in materials and workmanship for a period of 90 days. If you discover a defect, Parallax will, at its option, repair, replace, or refund the purchase price. Simply call for a Return Merchandise Authorization (RMA) number, write the number on the outside of the box and send it back to Parallax. Please include your name, telephone number, shipping address, and a description of the problem. We 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 this book, you find that it does not suit your needs, you may return it for a full refund. Parallax will refund the purchase price of the product, excluding shipping / handling costs. This does not apply if the book has been altered or damaged.

Copyrights and Trademarks

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, and any costs or recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax 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 microcontroller application, no matter how life threatening it may be.

Internet Access

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:	sxtech@parallaxinc.com
Web:	http://www.parallaxinc.com

Internet SX Discussion List

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

1 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 your first applications for the SX.

Development systems for the SX are offered by several vendors. In this introduction, we will use 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 indicates that a section contains useful additional information that is not necessary for understanding the current topic.

The "Tip" symbol marks a section that contains hints on using the development system, or other practical hints that might help you optimizing your work with the SX.

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 that topic can be found in the reference section of this book.

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. 8 bits. Instead of writing binary numbers, we will use two hexadecimal digits instead to represent the contents of a register in most cases. The SX program memory is addressed with 11 bits (SX18/28) or 12 bits (SX48/52). To represent an address value, 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 notation that most of the SX Assemblers expect:

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 typically used by C programmers.

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 systems using a new type of microcontroller, you usually will have 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

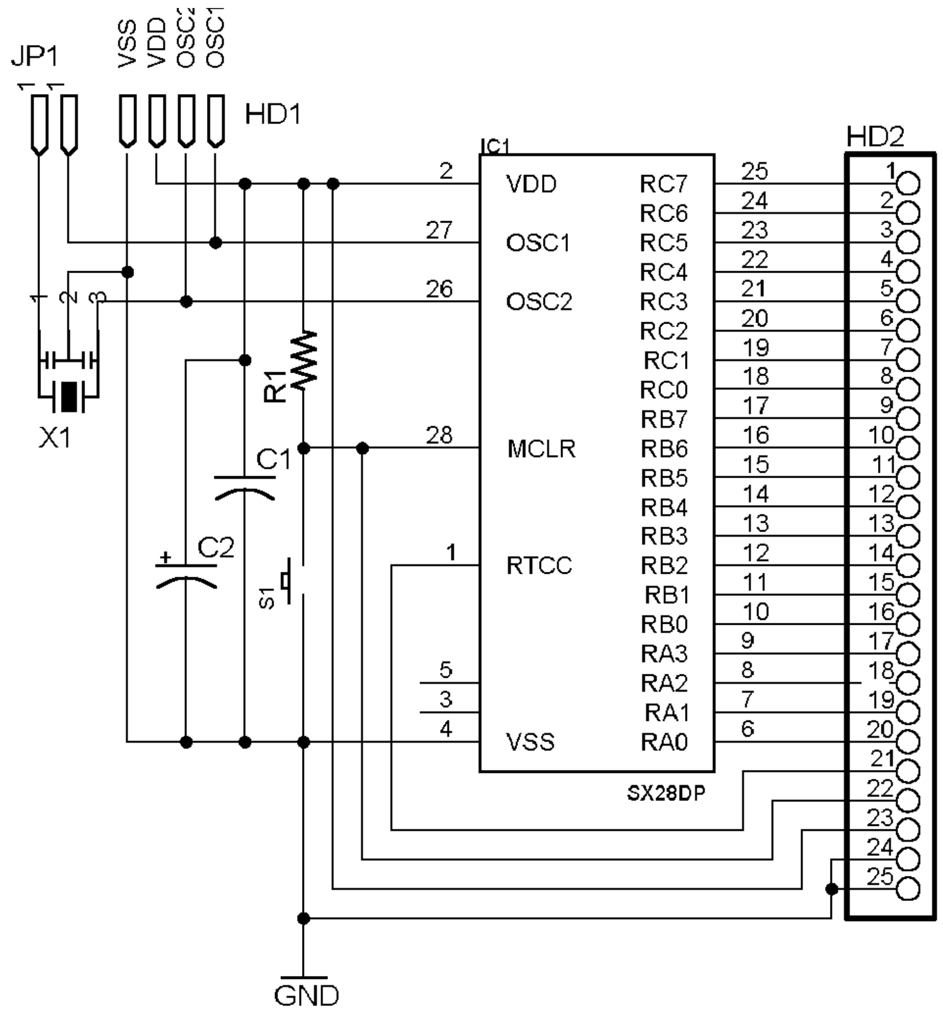
When you perform your first experiments with the SX, it is most likely that you will not yet have a finished PCB on hand designed for the SX system you intend to develop. Parallax, Ubicom, and other vendors offer various prototype boards.

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, 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 I^2C 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 using a schematic that should at least contain the following components:


JP1
VSS
VDD
OSC2
OSC1
HD1
IC1
HD2
VDD
OSC1
OSC2
MCLR
RTCC
VSS
RC7
RC6
RC5
RC4
RC3
RC2
RC1
RC0
RB7
RB6
RB5
RB4
RB3
RB2
RB1
RB0
RA3
RA2
RA1
RA0
SX28DP
X1
R1
C1
C2
S1
GND

SX 28 AC/DP

Component List

Name	Dimension	Remark
R1	10 kΩ	
C1	100 nF	Filter capacitors, use two or more if necessary.
C2	100 µF Tantalum	
S1		Reset button
X1	50 MHz Ceramic reso-nator	Alternatively you may use a 50 MHz crystal
	SX 28, DIP package	On a PCB, use a socket
JP1		Open when the development system is connected to HD1
HD1	4-pin header connector, 1/10’’ spacing	Connector for SX-Key
HD2	25-pin header connector	SX ports, RTCC, reset, and power supply

In order to connect external components to the SX, you should provide header pins for the port lines, for the RTCC input, the MCLR input, and the 5V stabilized power supply.

Take care that the leads connecting to OSC1 and OSC2 are as short as possible as the clock frequency may be up to 100 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 created SX-Key, a development system for the SX. The major component is a serial cable with a female 9-pin SUB-D connector to be connected to a serial COM port of a PC at one end, and with the other end connecting to a small PCB containing a 4-pole plug at the other end that connects to the 4-pin ISP header you can find on 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 the order that matches the one printed on the SX-Key plug, i.e. OSC1-OSC2-VDD-VSS. As this plug is not indexed, make sure that it is plugged in the right direction.

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

Together with the PC software that comes with the SX-Key you have a complete development system that allows you to write applications for the SX in Assembly language, program the SX, and test the application using the integrated debugger. All this is performed using the target SX on the prototype board. 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 the serial cable.

1.3.2 Installing the SX-Key IDE Software

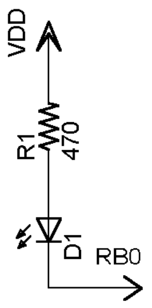
Together with the SX-Key, you should have received a diskette or CD-ROM with the SX-Key IDE software. You need a PC running a Windows-OS (Win 95 or greater). To install the software, run the setup program that is on the diskette or CD-ROM, or copy the contents of the disk into a directory of your hard-drive.

It makes sense to setup a shortcut to launch the SX-Key software (e.g. SXKey.exe) 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.parallaxinc.com) looking for newer versions of the SX-Key software. Parallax, Inc. offers new versions for download free of charge.

1.3.3 The First Program

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



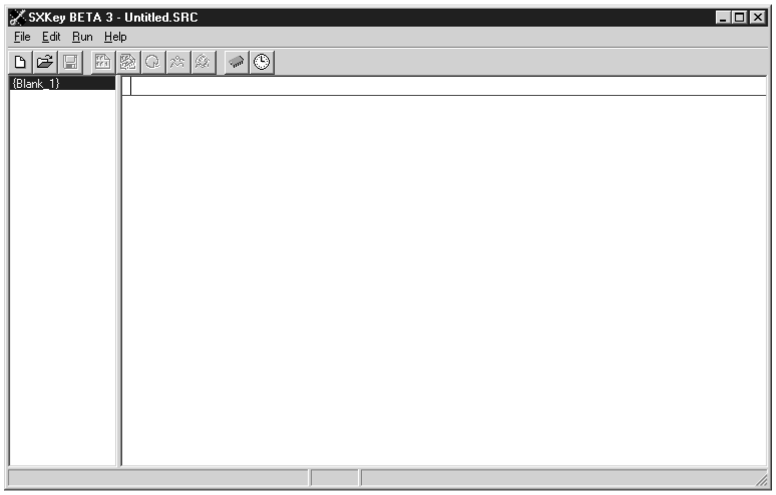
Some commercially available prototype boards do have an LED installed like this on the RB0 pin. On some boards, the LED may be connected between RB0 and V_{SS} instead. For our first tests, this does not matter.

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

Connect the SX-Key cable to a serial port of your PC and plug the other end to the ISP header pins (double-check the correct orientation of the plug). As the SX-Key cable is relatively short, you might consider using a serial extension cable. Make sure that you use a "straight-through" type of cable, i.e. not a null-modem cable. If you need adapters to convert between 9-pin and 25-pin DB connectors, make sure that the adapters are "straight-through". Finally, note whether the cable is connected to COM1, COM2, or another COM port.

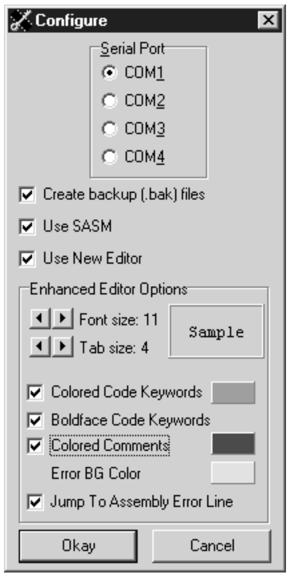
Make sure that the jumper that connects a resonator or crystal to the OSC1 pin on the prototype board is open. If there is no jumper, remove the resonator or crystal from its socket.

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

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


Configure

Serial Port: COM1 [✓], COM2, COM3, COM4

- Create backup (.bak) files ✓
- Use SASM ✓
- Use New Editor ✓

Enhanced Editor Options
- Font size: 11
- Tab size: 4
- Colored Code Keywords ✓
- Boldface Code Keywords ✓
- Colored Comments ✓
- Error BG Color
- Jump To Assembly Error Line ✓

Click the radio button next to the COM port you are going 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.

Now enter the following text in the editor window:

```
; TUT001.SRC
;
DEVICE   SX28L
DEVICE   TURBO, STACKX, OPTIONX
IRC_CAL  IRC_FAST
FREQ     50_000_000
RESET    0

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

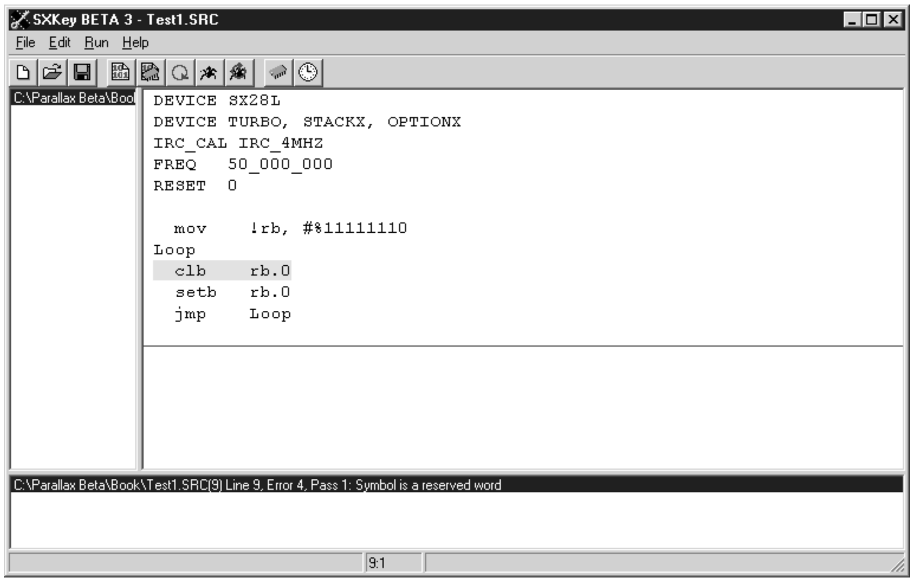
It makes no difference if you type uppercase or lowercase letters. You may enter tabs or spaces to separate the words. The leading spaces we have inserted in some lines is not actually required but they make the text look a bit more "structured". You may insert any number of empty lines to format the text.

This and the following sample programs assume that you are using an SX 28 controller. In case you use an SX 18, replace the DEVICE SX28L with DEVICE SX18L.

After you have entered your first program code, you need to save it. Either click on the diskette button, select “Save” from the “File” menu, or press Ctrl-S on the keyboard to open the “Save Source as...” Dialog. If you like, select the folder where the file shall be saved, or create a new one and then enter a file name, e.g. Test1 (the “.SRC“ extension is added automatically, so there is no need to enter it).

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 compile the little program you have entered before. If a dialog opens, telling you that the file needs to be saved prior to assembling, click the “Ok” button. You may consider selecting the “Don’t show this dialog again” option to avoid this dialog box in the future. When the status line at the bottom right of the editor window displays "Assembly Successful", the program was compiled without errors, and it can be executed.

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 like in the next picture.



```
DEVICE SX28L
DEVICE TURBO, STACKX, OPTIONX
IRC_CAL IRC_4MHZ
FREQ   50_000_000
RESET  0

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

```
C:\Parallax Beta\Book\Test1.SRC(9) Line 9, Error 4, Pass 1: Symbol is a reserved word
```

It is most likely that you have misspelled some text. The line, the assembler is "complaining" about is highlighted, and in the new section that has opened under the text, you can find a description of the error. In our example, the assembler error message reads, "Symbol is a reserved word". This may be a bit mis-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 mis-spelled, it should read "clrb".

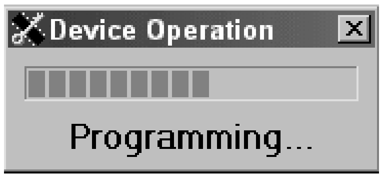
Make the necessary corrections to the text, and press Ctrl-A again to compile the modified program until the message "Assembly Successful" is displayed in the status line.

After correcting any errors, you should again save your "masterpiece". Click the Save button with the diskette symbol, select "Save" from the "File" menu, or press the shortcut Ctrl-S.

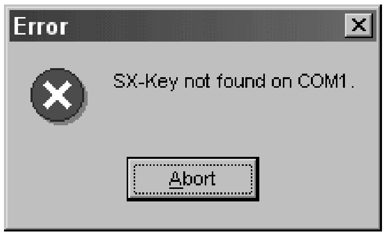
When source code files become larger, it is a good idea to save the file from time to time. Simply click the save button or press the Ctrl-S shortcut to make sure that your work is not lost. Please note that the editor does automatically generate a backup copy of the previously saved version of a file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

Now click the debug button (the fourth button from the right with the bug symbol), select "Debug" from the "Run" menu, or press Ctrl-D to launch the SX-Key debugger. Since the programs are always executed on the target SX controller, the program must be transferred to the SX first. Invoking the debugger means that the current version of the source code file is compiled, and then the program is transferred to the SX (provided that there are no errors in the source code).

When you see a display like



you are quite close to executing your first program. Should an error message come up like



SX-Key not found on COM1.

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

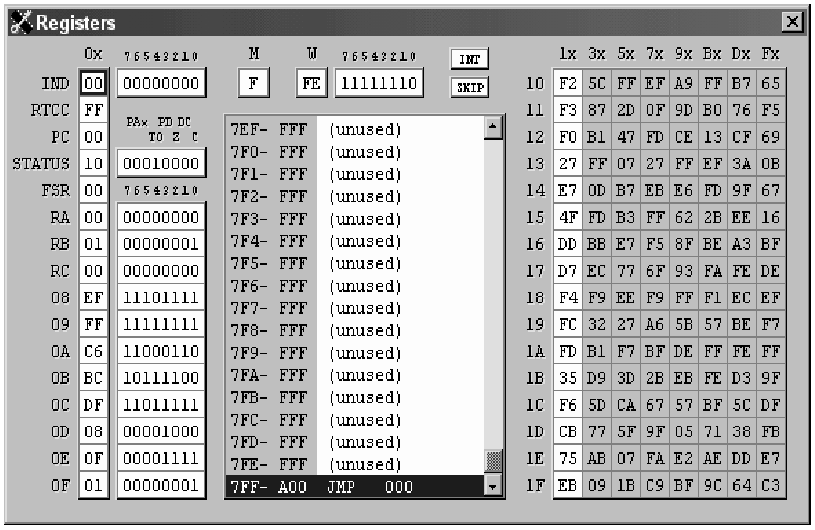
- No supply voltage connected to the prototype board, or supply voltage too low or too high.

- SX-Key cable not connected, or wrong orientation of the ISP plug.
- You have configured the wrong COM port.
- 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. You hopefully will have found the reason before reaching the last two items in the list.

After you have fixed the problem, press Ctrl-D again. After a few seconds, the "Programming" message should come up again. When the program has been transferred to the SX, the message box will be closed.

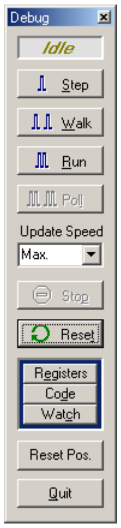
Next, the following windows will open:

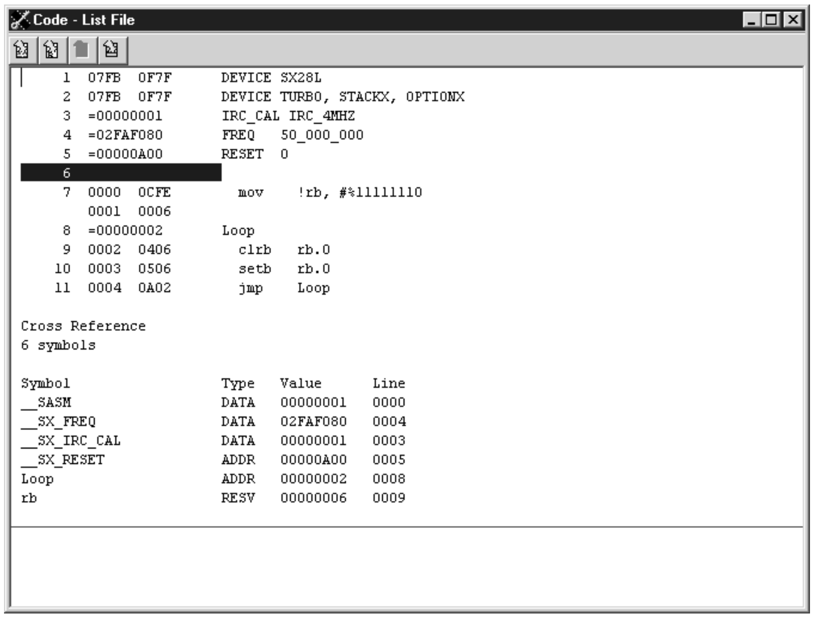


```
         0x   76543210       M      W    76543210     INT
IND      00   00000000       F      FE   11111110     SKIP
RTCC     FF
PC       00   PA× PD DC
                TO Z C
STATUS   10   00010000
FSR      00   76543210
RA       00   00000000
RB       01   00000001
RC       00   00000000
08       EF   11101111
09       FF   11111111
0A       C6   11000110
0B       BC   10111100
0C       DF   11011111
0D       08   00001000
0E       0F   00001111
0F       01   00000001
```

```
7EF-  FFF   (unused)
7F0-  FFF   (unused)
7F1-  FFF   (unused)
7F2-  FFF   (unused)
7F3-  FFF   (unused)
7F4-  FFF   (unused)
7F5-  FFF   (unused)
7F6-  FFF   (unused)
7F7-  FFF   (unused)
7F8-  FFF   (unused)
7F9-  FFF   (unused)
7FA-  FFF   (unused)
7FB-  FFF   (unused)
7FC-  FFF   (unused)
7FD-  FFF   (unused)
7FE-  FFF   (unused)
7FF-  A00   JMP    000
```

	1x	3x	5x	7x	9x	Bx	Dx	Fx
10	F2	5C	FF	EF	A9	FF	B7	65
11	F3	87	2D	0F	9D	B0	76	F5
12	F0	B1	47	FD	CE	13	CF	69
13	27	FF	07	27	FF	EF	3A	0B
14	E7	0D	B7	EB	E6	FD	9F	67
15	4F	FD	B3	FF	62	2B	EE	16
16	DD	BB	E7	F5	8F	BE	A3	BF
17	D7	EC	77	6F	93	FA	FE	DE
18	F4	F9	EE	F9	FF	F1	EC	EF
19	FC	32	27	A6	5B	57	BE	F7
1A	FD	B1	F7	BF	DE	FF	FE	FF
1B	35	D9	3D	2B	EB	FE	D3	9F
1C	F6	5D	CA	67	57	BF	5C	DF
1D	CB	77	5F	9F	05	71	38	FB
1E	75	AB	07	FA	E2	AE	DD	E7
1F	EB	09	1B	C9	BF	9C	64	C3





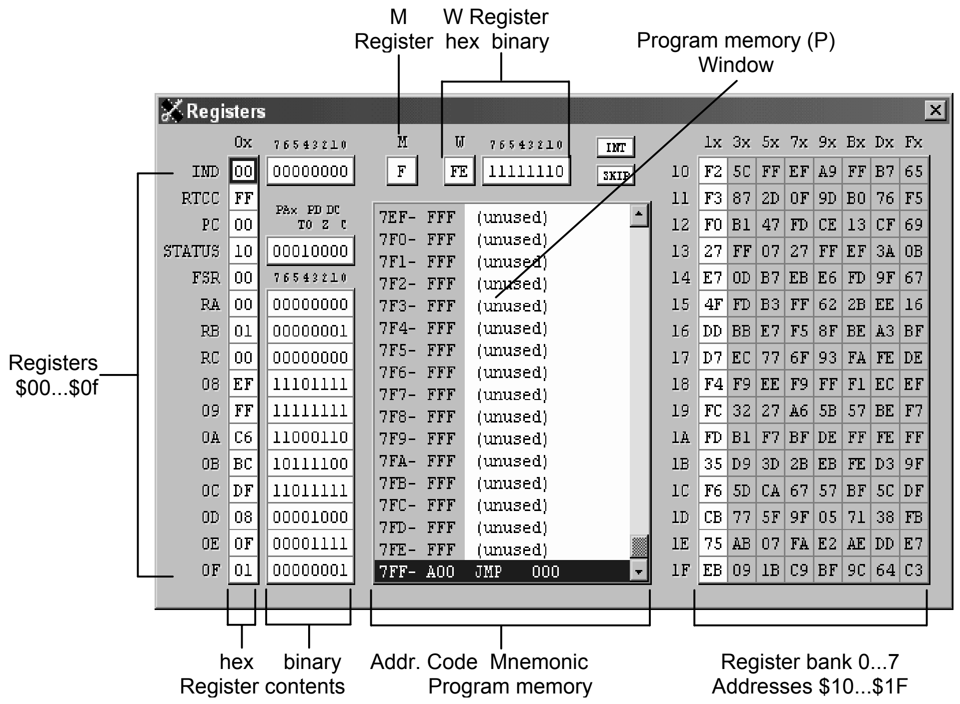
```
   1  07FB  0F7F       DEVICE SX28L
   2  07FB  0F7F       DEVICE TURBO, STACKX, OPTIONX
   3  =00000001        IRC_CAL IRC_4MHZ
   4  =02FAF080        FREQ   50_000_000
   5  =00000A00        RESET  0
   6
   7  0000  0CFE         mov    !rb, #%11111110
      0001  0006
   8  =00000002        Loop
   9  0002  0406         clrb   rb.0
  10  0003  0506         setb   rb.0
  11  0004  0A02         jmp    Loop

Cross Reference
6 symbols

Symbol                 Type   Value      Line
__SASM                 DATA   00000001   0000
__SX_FREQ              DATA   02FAF080   0004
__SX_IRC_CAL           DATA   00000001   0003
__SX_RESET             ADDR   00000A00   0005
Loop                   ADDR   00000002   0008
rb                     RESV   00000006   0009
```

The sizes and the positions of these windows depend on the screen resolution you have configured. You can move each of the windows, and the "Code - List File" window can be resized as well.

1.3.4 The SX-Key Debugger Windows

1.3.4.1 The Registers (R) Window

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


M Register
W Register hex binary
Program memory (P) Window
Registers $00...$0f
hex binary Register contents
Addr. Code Mnemonic Program memory
Register bank 0...7 Addresses $10...$1F
Registers
INT
SKIP
PAx PD DC
TO Z C

	0x	76543210
IND	00	00000000
RTCC	FF	
PC	00	
STATUS	10	00010000
FSR	00	
RA	00	00000000
RB	01	00000001
RC	00	00000000
08	EF	11101111
09	FF	11111111
0A	C6	11000110
0B	BC	10111100
0C	DF	11011111
0D	08	00001000
0E	0F	00001111
0F	01	00000001

M	W	76543210
F	FE	11111110

```
7EF- FFF  (unused)
7F0- FFF  (unused)
7F1- FFF  (unused)
7F2- FFF  (unused)
7F3- FFF  (unused)
7F4- FFF  (unused)
7F5- FFF  (unused)
7F6- FFF  (unused)
7F7- FFF  (unused)
7F8- FFF  (unused)
7F9- FFF  (unused)
7FA- FFF  (unused)
7FB- FFF  (unused)
7FC- FFF  (unused)
7FD- FFF  (unused)
7FE- FFF  (unused)
7FF- A00  JMP   000
```

	1x	3x	5x	7x	9x	Bx	Dx	Fx
10	F2	5C	FF	EF	A9	FF	B7	65
11	F3	87	2D	0F	9D	B0	76	F5
12	F0	B1	47	FD	CE	13	CF	69
13	27	FF	07	27	FF	EF	3A	0B
14	E7	0D	B7	EB	E6	FD	9F	67
15	4F	FD	B3	FF	62	2B	EE	16
16	DD	BB	E7	F5	8F	BE	A3	BF
17	D7	EC	77	6F	93	FA	FE	DE
18	F4	F9	EE	F9	FF	F1	EC	EF
19	FC	32	27	A6	5B	57	BE	F7
1A	FD	B1	F7	BF	DE	FF	FE	FF
1B	35	D9	3D	2B	EB	FE	D3	9F
1C	F6	5D	CA	67	57	BF	5C	DF
1D	CB	77	5F	9F	05	71	38	FB
1E	75	AB	07	FA	E2	AE	DD	E7
1F	EB	09	1B	C9	BF	9C	64	C3

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

1.3.4.2 The Code – List File (C) Window

This window displays the assembly source code as you have typed it in plus some additional information. Actually, this is the "List" file format that shows the machine codes the assembler has generated, and the addresses where the 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 buttons to open other 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).

1.3.5 Executing the First Program in Single Steps

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

Here the assembler has generated an instruction that makes the SX continue program execution at the address defined with the RESET directive in our program (0 in our example).

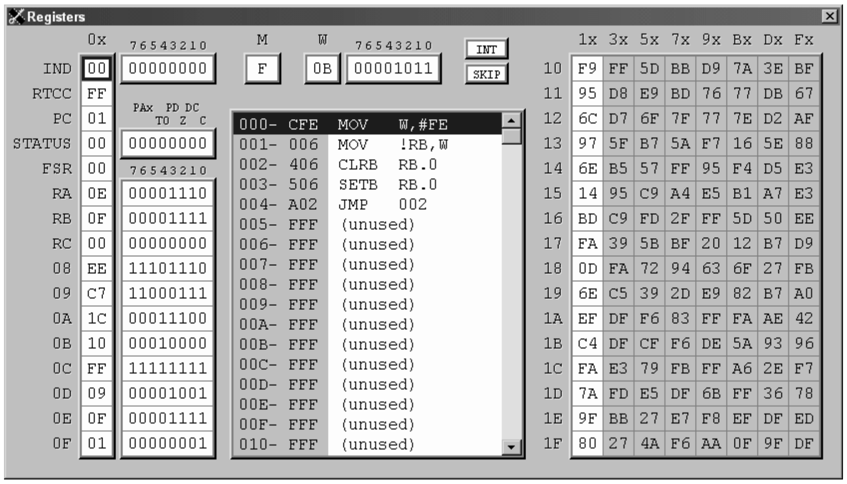
The instruction that unconditionally causes the program execution to branch to another location is the `jmp` instruction. This instruction loads the program counter (PC) with the target address, i.e. it then "points" to the next instruction to be executed.

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

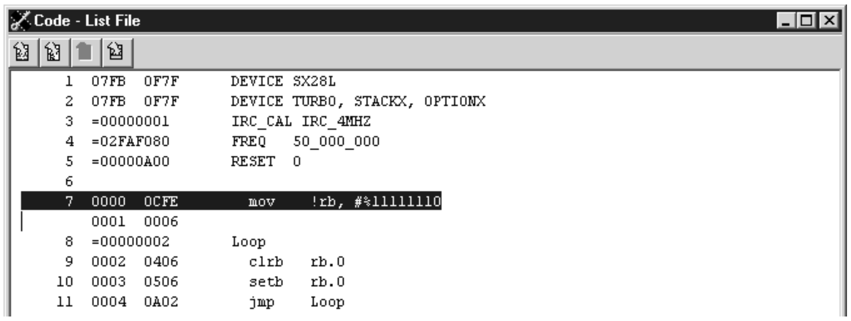
The same is true for the C window. Here you have horizontal and vertical scroll bars available to move the text.

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

Now the window contents have changed like this:



```
Registers
         0x  76543210      M    W   76543210   INT         1x 3x 5x 7x 9x Bx Dx Fx
   IND   00  00000000      F   0B   00001011   SKIP    10  F9 FF 5D BB D9 7A 3E BF
  RTCC   FF                                            11  95 D8 E9 BD 76 77 DB 67
    PC   01  PAx PD DC                                 12  6C D7 6F 7F 77 7E D2 AF
               TO Z  C  000- CFE  MOV   W,#FE
STATUS   00  00000000   001- 006  MOV   !RB,W          13  97 5F B7 5A F7 16 5E 88
   FSR   00  76543210   002- 406  CLRB  RB.0           14  6E B5 57 FF 95 F4 D5 E3
    RA   0E  00001110   003- 506  SETB  RB.0           15  14 95 C9 A4 E5 B1 A7 E3
                        004- A02  JMP   002
    RB   0F  00001111   005- FFF  (unused)             16  BD C9 FD 2F FF 5D 50 EE
    RC   00  00000000   006- FFF  (unused)             17  FA 39 5B BF 20 12 B7 D9
    08   EE  11101110   007- FFF  (unused)             18  0D FA 72 94 63 6F 27 FB
    09   C7  11000111   008- FFF  (unused)             19  6E C5 39 2D E9 82 B7 A0
                        009- FFF  (unused)
    0A   1C  00011100   00A- FFF  (unused)             1A  EF DF F6 83 FF FA AE 42
    0B   10  00010000   00B- FFF  (unused)             1B  C4 DF CF F6 DE 5A 93 96
    0C   FF  11111111   00C- FFF  (unused)             1C  FA E3 79 FB FF A6 2E F7
    0D   09  00001001   00D- FFF  (unused)             1D  7A FD E5 DF 6B FF 36 78
                        00E- FFF  (unused)
    0E   0F  00001111   00F- FFF  (unused)             1E  9F BB 27 E7 F8 EF DF ED
    0F   01  00000001   010- FFF  (unused)             1F  80 27 4A F6 AA 0F 9F DF
```



```
Code - List File
07FB  0F7F      DEVICE SX28L
07FB  0F7F      DEVICE TURBO, STACKX, OPTIONX
=00000001       IRC_CAL IRC_4MHZ
=02FAF080       FREQ   50_000_000
=00000A00       RESET  0

0000  0CFE        mov    !rb, #%11111110
0001  0006
=00000002       Loop
0002  0406        clrb   rb.0
0003  0506        setb   rb.0
0004  0A02        jmp    Loop
```

The highlight has moved to address $000, i.e. the SX has executed the **`jmp 0`** instruction it has found at $7ff, and this has caused a jump to the new program address $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 1, !rb, #%11111110`** instruction. The Assembler automatically generated two separate instructions that mean the same thing:

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

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

The hash-sign "#" to the left of the binary number %11111110 means that the constant value %11111110 shall be copied to a target called !rb. The hash-sign is very important here - if you leave it out, the assembler assumes 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 shall be configured as an output. To clarify which bits are set and cleared in the !rb register, we use binary notation here.

As you can see, in the P window, values are always displayed in hex (without the leading "$").

Here, we configure the RB0 pin as an output - this is where we have connected the LED.

Actually, the `mov !rb, #%11111110` is composed of the instructions

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

i.e. the constant value %11111110 is copied into the w register, and then the contents of the w register 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” register found in other microcontrollers or microprocessors.

Now click the "Step" button again, and you will see that the highlight in the P window now has moved to the second part of the compound instruction where the highlight in the C window did not move.

Click "Step" again to execute the `mov !rb, w` instruction, and to bring the highlight on 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 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 forget the leading "#" or "%" characters)?
- Do all instructions address the correct port (rb)?
- Do the `clrb rb.0` and `setb rb.0` instructions both contain the "0", and no other 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, click the "Quit" button to return to the editor, make the necessary corrections, and press 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 the power supply to 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.

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.

As you did not make any changes to the program, it is not necessary to transfer the program into the SX when you want to restart the debugger again. Therefore, don't select "Run - Debug" or don't press the Ctrl-D shortcut. Instead, select "Run - Debug (reenter)", or enter Ctrl-Alt-D. This starts up the debugger immediately without transferring the program into the SX again.

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

Now click the "Hop" button instead of the "Step" button. You will not notice any difference - the jump to address $000 is performed as before. Now click "Hop" again to see the difference: the highlight is positioned at address $002. This means that the SX has executed both instructions that make the compound `mov !rb, #%11111110` in "one step" (actually, it has performed both instructions in sequence at full speed).

Keep clicking the "Hop" button, and notice how the LED is turned on and off as you continue clicking the button.

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. Since the associated port pin has been configured as an output, this means that the output pin goes to a 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 to go to a 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. If you watch the contents of address $06 you will notice how the content changes as you keep clicking the "Hop" 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 shown with a red background when its state has changed. This helps you to quickly find out what data have changed after the execution of 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 has been executed.

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'll 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 insert an 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 forget to correct some 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 one of the next lines (`$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).

Note that some assemblers expect labels always beginning at the leftmost column of a line, although the SX-Assembler allows leading spaces. For compatibility reasons, it is a good idea to always let labels begin in the first column. This makes it easier when you later might want to compile a program with another assembler.

1.3.8 Running the program at full speed

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

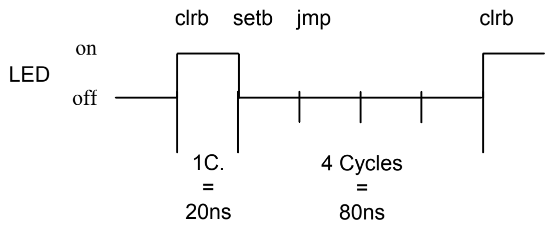
There are two reasons for that phenomenon: duty cycle and speed.

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 also 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 with a 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.

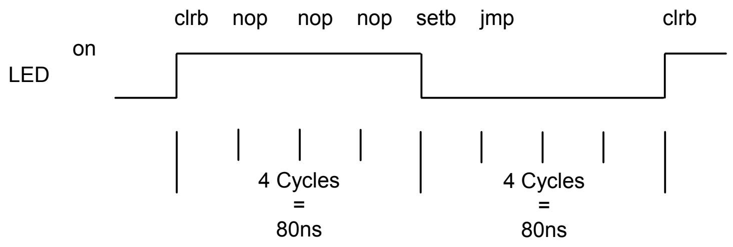
To extend the LED's on time, we need to add some "cycle eater" instructions between the `clrb` and `setb` instructions that "steal" three clock cycles. The SX "knows" a special "do nothing" instruction that does this, the `nop` (for no operation).

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
```

Since you changed the source code, you can't restart the debugger now with just the Debug Reenter feature, therefore press Ctrl-D to compile 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 built 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 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 that was executed when you clicked the button, and all registers reflect the status at that time, and you may continue program execution from that point.

Instead of "Walk", you can also click "Jog" to make the LED blink. In "Jog" mode, the debugger steadily clicks the "Hop" button for you, i.e. the instructions that make up compound instructions are executed at full speed in this mode.

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) instruction. This instruction decrements the specified register (at address $08 in the data memory here) by one. If the content of the register ends up at 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 is 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. Because 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. So adding the three more cycles for the final `jmp Loop` instruction, we finally end up in 1,023 * 2 + 3 = 2,047 cycles that take 2,047 * 20 ns = 40.94 µs which results in a 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:

```
; TUT002.SRC
;
DEVICE SX28L
DEVICE TURBO, STACKX, OPTIONX
IRC_CAL IRC_FAST
FREQ   50_000_000
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
```

Here, 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:

Since 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 * 20ns ≈ 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. Demo2.src), and then press Ctrl-D to launch the debugger.

Click the "Run" button to execute the program. Provided 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 "picture" of the current register status, click the "Poll" button at any time the program is executed at full speed.

If 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 "Jog" or "Walk" modes take 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 single-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 currently pointing to up to and including the instruction where you set the breakpoint.

To set a breakpoint, first make sure that the program is halted by either clicking the "Stop" or "Reset" button. 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 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 again after executing the `clrb` instruction.

Please note that there can only be one breakpoint active at a time. 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 normal.

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.

At the top of the top of the Code – List File window, you notice four buttons that are handy to position the cursor at specific points:

From left to right, the buttons have the following meaning:

Go to Code	- This positions the cursor at the first line with executable code, i.e. all initial comments, definitions, etc. are skipped.
Go to Reset	- This positions the cursor at the first line of the code that will be executed after a reset, i.e. the line that is defined by the RESET directive.
Go to Breakpoint	- This positions the cursor at the code line that is marked with a breakpoint (if no breakpoint is set, this button will be inactive).
Go to next Run Line	- This positions the cursor at the code line that will be executed next when the debugger is in single-step mode.

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.

1.4 SX Configuration - ORG/DS - Conditional Branches

1.4.1 Configuration Directives

In the sample programs shown before as well as in all following programs, there is a section at the beginning, similar to:

```
DEVICE  SX28L
DEVICE  TURBO, STACKX, OPTIONX
IRC_CAL IRC_FAST
FREQ    50_000_000
RESET   0
```

Statements like this are called "Assembler Directives". They do not cause the assembler to generate program code. Instead, they instruct the development system how to configure the SX when it transfers a program to the microcontroller. For configuration purposes, the SX has special registers called "Fuse Registers".

The `DEVICE` directives make it easy to define the settings of the fuse register bits because you need not to remember which fuse bit is used to set a specific option like the turbo mode or an extended stack and option register.

(2.2.8.1 - 234) The parameters following a `DEVICE` directive each define a specific configuration. For example, `SX28L` informs the assembler that the generated program shall be transferred into an SX28 device. `TURBO` activates the turbo mode, i.e. each standard instruction will be executed in one clock cycle, and `STACKX` and `OPTIONX` activate an 8-level return stack for subroutines and an extended option register.

You may wonder why the SX offers the “non-turbo” mode, where each standard instruction is executed in four clock cycles, the smaller 2-level return stack, and the reduced option register functionality. This mode of operation is similar to some similar microcontroller devices manufactured by other Vendors. In “real life”, you will never use this reduced functionality with an SX. Why would you drive a Porsche in the to lowest gears only?

As you can see, a `DEVICE` directive can be followed by more than one parameter. Use commas to separate `DEVICE` parameters.

The `IRC_CAL` and `FREQ` directives are internal for the SX-Key debugger. They specifiy the calibration of the SX-internal clock oscillator and the clock frequency, in Hz, the debugger shall generate when the SX runs at full speed. Note that you may insert underscores for better readability of the frequency parameter.

We have already described the meaning of the `RESET` directive. It informs the assembler about the starting address of the main program, and the assembler generates a `jmp` instruction to that address in the highest location of program memory.

Note that the SX-Key assembler and other assemblers like SASM use a slightly different syntax for the `DEVICE` directives.

From now on, we will start each program with a configuration section like this:

```
ifdef __SASM
  DEVICE  SX28L, STACKX, OPTIONX
  IRC_CAL IRC_FAST
  FREQ    50_000_000
else
  DEVICE  SX28AC, OSCHS, OPTIONX
endif
DEVICE  TURBO
RESET   Start
```

Here, we use a construct called "Conditional Assembly" that we will be discussing in detail later in this tutorial. For now, just keep in mind that the directives between `ifdef __SASM` and `else` will be respected as long as the symbol `__SASM` is defined, and the directives between `else` and `endif` will be ignored. This symbol is defined by default when you use the SASM assembler that comes with the new SX-Key Software.

When you compile a program with some assembler other than the SX-Key software’s integrated SASM assembler, this symbol is not defined by default, and the assembler will ignore the directives between `ifdef __SASM` and `else` and it will respect the ones between `else` and `endif`.

If you are using a development system other than the SX-Key, make sure that the SX is clocked at 50 MHz because many of the sample programs assume this clock speed when generating timing delays, interrupts, etc.

1.4.2 The ORG and DS directives

In the previous program, we have used three nested loops to "slow down" the SX in order to see the LED blink. To build a loop that is executed a certain number of times, you need a loop counter that is incremented or decremented (as in our example) until a specific value has been reached (0 in our case). We have used registers at address $08, $09 and $0a in the data memory as the loop counters in this example.

Let's re-write this example and make it a bit more "generic":

```
; TUT003.SRC
;
ifdef __SASM
  DEVICE  SX28L, STACKX, OPTIONX
  IRC_CAL IRC_FAST
  FREQ    50_000_000
else
  DEVICE  SX28AC, OSCHS, OPTIONX
endif
DEVICE  TURBO
RESET   Start

  org $08
```

```
Counter1 ds 1
Counter2 ds 1
Counter3 ds 1

  org $100

Start
  mov    !rb, #%11111110
Loop
  decsz Counter1
    jmp Loop
  decsz Counter2
    jmp Loop
  decsz Counter3
    jmp Loop
  clrb   rb.0

Loop1
  decsz Counter1
    jmp Loop1
  decsz Counter2
    jmp Loop1
  decsz Counter3
    jmp Loop1
  setb   rb.0

  jmp    Loop
```

Following the configuration directives, you can see the `org $08` ("originate") directive. This informs the assembler that definitions following this directive shall begin at address $08 (in data memory in this case).

In the next line, you find the statement `Counter1 ds 1`. As mentioned before, the assembler interprets a word that has no pre-defined meaning - like "Counter1" in this case - as a label and adds the word together with a numeric value that represents its address to the symbol table. The address of the label `Counter1` is $08 because we have instructed the assembler to continue counting from that address with the previous `org $08` directive.

The `ds` following the label name means "define space", and the `1` following `ds` instructs the assembler to "set aside" one byte for `Counter1`. This also increments the assembler's internal address-counter by one. Therefore, the next label, `Counter2` is located at address $09. Because `ds 1` also reserves one byte for `Counter2`, `Counter3` is located at address $0a.

You can think of the three reserved bytes as three variables, each having a size of one byte, named `Counter1`, `Counter2` and `Counter3`.

Within the loops, you will notice that the `decsz` instructions no longer refer to "hard-coded" addresses in data memory, but use the variable names instead.

Using symbolic addresses or names for variables makes a program much more readable, and it is a lot easier to modify the program later.

Note that there is another `org` directive in the program: `org $100`. As before, it instructs the assembler to set its internal address pointer to the specified value ($100 in our example).

The next instruction, `mov !rb, #%11111110` will be coded into that address. This means that our program no longer begins at address $000 in the program memory but at address $100 instead.

Since we have placed the label `Start` in front of the first instruction, we can use that label to specify that address in the `RESET` directive.

Note that one of the `org` directives specifies an address in data memory, where the other one specifies an address in program memory. The assembler "sorts that out" automatically. Also note that in case of the `Start` label, we have a "forward declaration", i.e. the label is referred to in the source code before it is actually defined. Again, the assembler takes care of this (it actually does an extra run through the source code where it "collects" all the label definitions before compiling the instruction code).

1.4.3 Conditional Branches

In the sample programs above we have already used the `decsz` instruction to build the delay loops. Let's discuss such kinds of instructions in more detail now.

Without the capability to change the flow of program execution depending on certain conditions, a microcontroller would be rather useless as it could only execute "straight through" types of programs. Therefore, the SX comes with a set of conditional skip instructions, like the `decsz` instruction.

```
decsz Counter1
  jmp Loop
```

This example decrements the content of `Counter1`, i.e. it subtracts one from the former content, and the result becomes the new content of `Counter1`. In case that the result yields zero, the instruction immediately following the `decsz` is skipped (the `jmp Loop` in our example).

If you know microprocessors or other controllers, you may wonder why the SX does not "know" instructions JZ or JNZ (Jump if Zero or Jump if Not Zero). This is because each instruction code always consists of just one word, 12 bits wide, but 12 bits are not enough to hold a register address as well as a jump address.

Therefore, keep in mind that the instruction following a conditional skip is not executed when the condition is true.

You may wonder why we have indented the `jmp` instructions following the `decsz` instructions. The first reason is to make the program more readable, and the indentation indicates that this instruction is not always executed. The second reason is much more important.

Look at the following code snippet:

```
decsz Counter1
  mov !rb, #%11111110
```

On the first glance, this sequence seems to be fine, but it contains a "ticking bomb"! Remember that the `mov !rb, #%11111110` is a compound instruction as the SX does not provide a basic instruction that can copy a constant value directly into a port configuration register. The assembler compiles the instructions like this:

```
decsz Counter1
  mov w, #%11111110
  mov !rb, w
```

You can now immediately see what the problem is: The `decsz` instruction does not "know" about compound instructions. All it does is increment the PC register in order to skip the next instruction in case the condition is true. In our example, the `mov w, #%11111110` instruction will be skipped but not the `mov !rb, w` instruction! The result is that `!rb` will receive a random value, depending on the current content of `w`, and this is definitely not what you want.

Therefore, the instruction that immediately follows a skip instruction *must never* be a compound instruction; otherwise, strange results may occur.

There are two ways to avoid that dangerous situation: Not using compound statements at all, or at least paying special attention to it.

Not using compound statements at all is possible because you can always write the basic instructions that make a compound instruction, but this means additional typing work, and increases the size of the source code.

Indenting the instruction following a skip makes it easier to double-check for not using compound instructions at such places.

1.5 Subroutines - Symbols - Data Memory

1.5.1 Subroutines

If you look at the previous LED-Blinker program, you will notice that the nested delay loops are duplicated, i.e. one is executed before turning the LED on, and the other is executed before the LED is turned off again.

Subroutines can help to avoid such duplications, as shown in the following program version:

```
; TUT004.SRC
;
ifdef __SASM
  DEVICE  SX28L, STACKX, OPTIONX
  IRC_CAL IRC_FAST
  FREQ    50_000_000
else
  DEVICE  SX28AC, OSCHS, OPTIONX
endif
DEVICE  TURBO
RESET   Start

org $08
Counter1 ds 1
Counter2 ds 1
Counter3 ds 1

org $000
TimeEater
Loop1
  decsz Counter1
    jmp Loop1
  mov Counter1, #50
  decsz Counter2
    jmp Loop1
  decsz Counter3
    jmp Loop1

  ret

org $100
Start
  mov    !rb, #%11111110
Loop
  call TimeEater
  clrb   rb.0
  call TimeEater
  setb   rb.0
  jmp    Loop
```

Here we have moved the delay loops to a subroutine called `TimeEater`. A subroutine is a sequence of instructions terminated with a `ret` (Return) instruction.

To execute the instructions in a subroutine, you use the `call` instruction together with the address where the first instruction of the subroutine is located. In our example, we have defined the label `TimeEater` as a symbolic address for the subroutine entry, and therefore we used that label together with the `call` instructions.

Similar to a `jmp` instruction, a `call` unconditionally causes a branch to the specified address, i.e. the content of PC is changed accordingly.

As soon as the `ret` instruction is reached, the content of PC is restored to point to the address of the instruction immediately following the `call` which caused the branch.

As you can see, in the program there are two call instructions, both invoking the `TimeEater` subroutine.

Within `TimeEater` you find the nested delay loops with the extension that `Counter1` is now initialized to 50 when it underflows. This makes the delay a bit shorter in order to increase the LED's blink frequency.

Instead of duplicating the delay loops twice in the program, we have now just one set of delay loops in the subroutine, and we call the subroutine twice instead.

In this example, this only saves a few words in program memory but you can imagine that subroutines help to save a remarkable amount of program memory space.

Besides this, subroutines also help structure a program. Think of subroutines as "black boxes" that perform a specific task. So the calling program needs not to take care of the details, it just calls the subroutines, relying that the subroutines do their job properly.

1.5.2 The Stack

(2.2.7 - 230) Previously, we had mentioned that a `ret` instruction terminates a subroutine, and that it restores the PC register to point at the instruction following the call. This means that the content of PC+1 as return address must be saved somewhere before loading it with the entry-address of the subroutine.

If there were just one fixed register to save the return address, it would not be possible to call another subroutine from within a subroutine, although this is common programming practice.

The second call of a subroutine would overwrite the previously saved value, i.e. the return address for the first-level subroutine call would get lost.

Therefore, a memory structure called "Stack" is used so save a certain number of return addresses to allow for nested subroutine calls. A stack structure is also called LIFO (Last In First Out) as this describes the way data can be stored and retrieved.

You can think of a LIFO as a stack of dishes. If you put a new dish on the stack, it is a good idea to later remove the top dish first in order to avoid a disaster.

The SX has a stack that can hold up to eight return addresses (in "compatibility mode", the SX18/20/28 can only hold two addresses). This means that the maximum nesting depth for subroutines is eight. If you'd like to see what happens when the depth is exceeded, single-step through the following program:

```
; TUT005.SRC
;
ifdef __SASM
  DEVICE  SX28L, STACKX, OPTIONX
  IRC_CAL IRC_FAST
  FREQ    50_000_000
else
  DEVICE  SX28AC, OSCHS, OPTIONX
endif
DEVICE  TURBO
RESET   Start
org     $000
sr1
  call sr2
  ret
sr2
  call sr3
  ret
sr3
  call sr4
  ret
sr4
  call sr5
  ret
sr5
  call sr6
  ret
sr6
  call sr7
  ret
sr7
  call sr8
  ret
sr8
  call sr9
  ret
sr9
  ret
org     $100
Start
  call sr1
  jmp  Start
```

In the SX, the stack is dedicated to storing return addresses only. It cannot be used to store other data, and there are no PUSH or POP instructions available that you may know from other microprocessors.

This is not possible because in the SX, data and program code is stored in different memory, having different size.

In order to make use of the 8-level stack, make sure that the SX is configured accordingly, i.e. include the `DEVICE STACKX` directive (`DEVICE STACKX_OPTIONX` for the SX-Key Assembler) directive.

1.5.3 Local Labels

In the previous LED-Blinker sample program, we have used the label `Loop` to name the main program loop, and the label `Loop1` to name the delay loop within the subroutine.

See what happens when you rename the delay loop in the subroutine to `Loop` as well:

```
TimeEater
Loop
    decsz Counter1
      jmp Loop
    mov Counter1, #50
    decsz Counter2
      jmp Loop
    decsz Counter3
      jmp Loop
```

When you try to assemble this modified program, you will get an error message like "Label is already defined". Obviously, it is not possible that two labels representing different addresses can have the same name, otherwise the assembler would not "know" which address it should use when the program contains a reference to a label.

On the other hand, when programs become larger, you must be creative to "invent" new label names that are not only unique but also describe the meaning of what the labels stand for.

If you think of subroutines that are "generic" enough to be used in different programs, you must even take care that a subroutine copied to, or included with another source code does not make use of labels that are defined elsewhere in the code.

Fortunately, most assemblers for the SX allow the definition of local labels. Modify the incorrect code sample above to look like

```
TimeEater
:Loop
    decsz Counter1
      jmp :Loop
    mov Counter1, #50
    decsz Counter2
      jmp :Loop
    decsz Counter3
      jmp :Loop
;
;...
;
Start
  mov    !rb, #%11111110
```

```
:Loop
  call TimeEater
  clrb   rb.0
  call TimeEater
  setb   rb.0
  jmp    :Loop
```

and try to assemble the program again. This time, the assembler does not complain, and the program runs as expected although the subroutine and the main program loop make use of labels named `:Loop`.

The leading colon in front of the label names make `:Loop` local labels. A local label is valid only in the area of a program that is enclosed by two non-local, i.e. public labels.

This means that the first `:Loop` label in our example is valid from `TimeEater` up to `Start`, and the second `:Loop` label is valid from `Start` through the rest of the program code, where `TimeEater` and `Start` are both public labels.

Tip

This tip helps you avoid unnecessary headaches:

Just "for fun", insert a new global label `Foo` in `TimeEater`, following the first `jmp :Loop` instruction:

```
TimeEater
:Loop
  decsz Counter1
    jmp :Loop
Foo
  mov Counter1, #50
  decsz Counter2
    jmp :Loop
```

and try to compile the program. This time, the assembler will report that the label `:Loop` in the line following the `decsz Counter2` instruction is not defined.

According to the definition of the range in which a local label is valid, this is correct because the `:Loop` label following `decsz Counter2` is now only valid between `Foo` and `Start`, and there is no definition for `:Label` in this area.

Here, in this small program, such problems can be localized quite easily but imagine how difficult this might be in a large program with many local labels. Therefore, pay special attention when inserting new global labels "in the middle" of a program.

When the assembler comes across a global label, it is stored in the symbol table, but also in a "Recent Global Label" buffer.

When the assembler comes across a local label, it builds the full label name by appending the local label name to the recent global label, and stores this name in the symbol table. For example, when the recent global label is `Subroutine`, a local label `:NoClear` would be stored as `Subroutine:NoClear` in the symbol table.

When the assembler comes across the reference to a local label, it first builds the full name from the recent global symbol and the local label, before searching it in the symbol table. On the other hand, the assembler also accepts references to full names.

This means that you can refer a specific local label even from a location that follows a new global label. However, in this case, you must specify the label in full, like in the following example:

```
org    $000

Subroutine
  clr  w
:NoClear
  mov  $09, w
  ret

org    $100

Main
  call Subroutine
  call Subroutine:NoClear
  call :NoClear                ; This causes an error
```

Although this code is of no specific use, it demonstrates how two different entry points of a subroutine can be accessed from the main program. First, the "regular" call enters the subroutine at the main entry point that causes `w` to be cleared, using the global name `Subroutine`. The second call refers to a local label within the subroutine by specifying the "full" label name (`Subroutine:NoClear`) i.e. `w` is not cleared. The third call causes an error because the assembler tries to locate the label `Main:NoClear` which does not exist.

1.5.4 Some More Considerations about Subroutines

You may have wondered why we have originated the `TimeEater` subroutine at address $000, and why the main program begins at $100 now.

To understand this, it is important to know that the call instruction only contains eight bits to specify the address of a subroutine. This is how the instruction code for call is structured:

```
1001 aaaa aaaa
```

When the assembler compiles a `call` instruction, it replaces the lower eight bits `aaaa aaaa` in the instruction code with the lower eight bits of the address that is specified together with the call instruction. As a result, the entry point of a subroutine can only be specified as values between $000 and $0ff.

Therefore, it makes sense to reserve the lower area in program memory from $000 through $0ff for subroutines, and let the main program begin at $100.

Later we will see that address $000 has a special meaning when we discuss interrupts.

When the program memory from $000...$0ff is not large enough to hold the code for all of your subroutines, there are two options:

1. Usage of another program memory page (we will discuss this later in the tutorial).

2. Jump to a higher address in program memory - here comes an example:

```
RESET     Start
org       $000

SR1
  jmp     _SR1

org       $100

Start

; Initializations

:Loop

  ; Instructions within the main loop

  call    SR1
  jmp     :Loop

_SR1

  ; Instructions within the subroutine

ret
```

As you can see, the subroutine `SR1` requires one word only in the program area from $000 to $0ff for the `jmp _SR1` instruction where the remaining instructions for the subroutine are located behind the main program loop in memory above $100. The only little disadvantage is the additional word required for the `jmp` instruction and the three extra clock cycles needed to execute the `jmp`.

Fortunately, the assemblers report an error when you try to compile a program where the entry point of a subroutine lies above $0ff (this may happen when you add more instructions to a subroutine that cause following subroutines being "pushed up" in program memory. Error messages will be similar to "Address is not within lower half of memory page" or "ERROR: CALL must be to first half of page".

As mentioned before, a subroutine should behave like a "black box", i.e. it should be possible to call it from any location in the program, without any side effects caused by the subroutine. This especially means that the subroutine may not make changes to register contents that are important for other parts of the program.

For example, a function in the C programming language can fulfill this requirement when all arguments are passed to the function by value, and when the function uses local variables only. In addition, a function can optionally return a value to the calling program.

Requirements like this cannot be realized with the SX that easy. The main reason is the lack of a stack for data that C uses for passing arguments and storing local variables.

When you write subroutines, take special care that no register contents are changed that are required to remain unchanged by the calling program. On the other hand, it is almost impossible to write a subroutine that does not modify the content of the `w` register. Therefore, the calling program should never trust that `w` remains unchanged after a subroutine call. The same is true for the flags in the status register that we will address later. Often, the `w` register, or the status registers are used to return a result from the subroutine to the calling program. Here is an example:

```
; The subroutine multiplies the contents of number by three
; and returns the result in w.
;
TimesThree
  mov w, Number
  add w, Number
  add w, Number
ret


mov  Number, #2
call TimesThree

; w now holds 6
```

Note that this simple subroutine does not handle results greater than 255.

A good method to protect variables from being overwritten by a subroutine is the usage of different memory banks that are dedicated to the subroutines and the main program. We will discuss this in the next chapter.

Sometimes, subroutines need to store intermediate results. In such cases, it is a good idea to reserve one or more variables that can be used for temporary storage. By convention, the contents of such variables are only valid from the point where a subroutine has saved a value there, until it terminates. This means that no other part of the program may rely on the variable's contents. Nor can any subroutine assume that the variables hold specific values when it is called next.

When you write programs with nested subroutine calls, take care that a subroutine at a lower level does not make use of the same temporary variables. This would make its contents invalid for the calling subroutine.

Ubicom recommends the definition of variables named `localTemp0`, `localTemp1`, `localTemp2`, etc. as temporary storage. The subroutine at the highest nesting level may use `localTemp0`, the subroutine at the next lower level may use `localTemp1`, etc.

Even when you make use of clearly defined conventions how to use variables, it is always a good idea to double-check the integrity of the variable usage, and it is also helpful to add comments to each subroutine to explain which variables are changed by the subroutines, and on with variables the subroutines rely.

1.5.5 Correctly Addressing the SX Data Memory

(2.2.2 - 204) The reference section in this book describes how the data memory is divided in eight memory banks of 32 bytes each (SX 18/20/28), where the first 16 bytes always lay in the first bank (bank 0), no matter which bank is currently active. Within this first bank, only the upper eight bytes can be used as general-purpose registers where the lower eight registers stand for the SX'es special registers.

Let's try to learn more about memory banks with this little program (this version contains a bug as you'll see):

```
; TUT006.SRC
;
ifdef __SASM
  DEVICE  SX28L, STACKX, OPTIONX
  IRC_CAL IRC_FAST
  FREQ    50_000_000
else
  DEVICE  SX28AC, OSCHS, OPTIONX
endif
DEVICE  TURBO
RESET   Start

Start
  inc $10
  inc $30
  inc $50
  inc $70
  jmp Start
```

Enter and assemble this program, and then step through it with the debugger. After the first step, the `inc $10` instruction will be executed that increments the memory location at $10. As you step through this instruction, watch the displayed content of $10, and see how it increments as expected.

Now, when you step through the `inc $30` instruction you might expect that the contents of memory location $30 will increment, but this is not the case. Instead, $10 is incremented again. The same happens when you step through the next two instructions. Each time, $10 is incremented, but not $50 or $70.

The problem here is that the instruction code for an `inc` only provides five bits for an address in data memory:

```
0010 101f ffff
```

When the assembler compiles the instruction, it replaces the five bits that are marked with "f" with the lower five bits of the address argument that is part of the inc instruction.

Using our example, the following codes result:

```
$10                ->         0001 0000
inc                ->    0010 101f ffff
Instruction code   ->    0010 1011 0000

$30                ->         0011 0000
inc                ->    0010 101f ffff
Instruction code   ->    0010 1011 0000

$50                ->         0101 0000
inc                ->    0010 101f ffff
Instruction code   ->    0010 1011 0000

$70                ->         0111 0000
inc                ->    0010 101f ffff
Instruction code   ->    0010 1011 0000
```

As you can see, the instruction code is always the same, this is why each of the `inc` instructions addresses $10, and not the other registers as you might have expected.

You may compare the organization of the SX data memory with a parking garage. Each parking deck has 32 lots, numbered from 0 to 31 where the lots 0 to 15 are reserved for special people, while the remaining lots from 16 to 31 are open to the public with the exception that parking is allowed for white cars of brand XYZ, type A only.

Let's assume, you have borrowed a car from a friend (of course, a white XYZ, type A), and you have arranged that you will return his car to our (fancy) parking garage.

Later, you park the car in deck 5, lot 16 and call your friend: "Thanks for the car, I have parked it in lot 16, you may pick it up there again".

You can be sure, your friend will get mad at you because you forgot to tell him in which deck you have left his car. How should he find his car when in all lots No. 16 in all decks there are white XYZ A-Types only?

This is similar to the SX data memory: For the upper 16 bytes in each bank, it is not enough to specify the address because this would always be a value from $10 to $1f. In addition, you must tell the SX which memory bank it shall use.

Now let's improve our program:

```
; TUT007.SRC
;
ifdef __SASM
  DEVICE  SX28L, STACKX, OPTIONX
  IRC_CAL IRC_FAST
  FREQ    50_000_000
else
  DEVICE  SX28AC, OSCHS, OPTIONX
endif
DEVICE  TURBO
RESET   Start
```