SX Microcontroller and SX-Key - Frequently Asked Questions (FAQ)

First Edition: May 05, 2005
Second Edition: June 01, 2005

Preface

This PDF document contains a compilation of topics found in the Parallax Forum's section dedicated to the SX microcontrollers, and of questions asked by customers when calling for SX support which might be of general interest for SX developers.

This will document will regularly be updated when new topics arise. Beginning with the second edition, new topics in an edition will be marked with "N<ed>" where <ed> is the edition number. Updated topics will be marked "U<ed>".

Parallax Inc. and the author, Günther Daubach do not guarantee for completeness or correctness of this document, and both are not responsible or liable for any damages incurred while using the information contained herein.

The topics in the document are sorted in more-or-less alphabetical order. Use the bookmarks, or the search function of you reader to find the topic you are looking for.

Happy SX Programming,

Günther Daubach

The / Modifier (and others, like ++, --, >>, <<, <>)

Q: What meaning has the "/" when used in instructions like

```
mov w, /$0C
```

A: The "/" in this context is used to modify the standard `mov w, fr` instruction. With the slash before the register address, the register value is read, inverted, and the result stored in the w register without changing the value in the register itself. For example after these instructions:

```
mov $0C, #%10101010
mov w, /$0C
```

the w register contains %01010101. This is similar to the instruction sequence

```
mov $0C, #%10101010
not $0C
mov w, $0C
```

but in this case, the value in $0C would be changed, i.e. inverted.

Note that there are other modifiers for the standard `mov w, fr` instruction, like ++, --, >>, <<, and <> which increment, decrement, rotate right through carry, rotate left through carry, or swap, i.e. exchange the upper four against the lower four bits before storing the result in w. In all cases, the value in the register will not be modified.

The >> Operator

Q: What is the meaning of the >> operator?

A: The >> operator, similar to other operators, like +, -, *, etc. can be part of an expression that is evaluated by SASM during assembly time, i.e. not when the program is executed by an SX device.

A practical use of the >> operator is shown in the code snippet, below:

```
mov m, #TextTable >> 8
mov w, #Hello
```

This prepares the m and w registers to point to a location in program memory that contains a value to be read with an IREAD instruction. Locations in program memory are addressed by 12 bits, where the upper four address bits must be contained in the lower four bits of the m register, and the lower eight bits in the w register, before executing an IREAD instruction.

The expression `#TextTable >> 8` takes the full 12-bit address of the symbol TextTable, and shifts it right by eight bits. The result contains the upper four bits of the original value in its lower four bits which is then stored in the m register as a constant value. This conversion is performed when the program is assembled to achieve the correct initialization for the m register. At run-time, the SX does not “know” about this – it simply initializes m with the correct value, determined by the assembler, and written to the instruction code.

Please note that the SX instruction set also contains a special version of the mov w, fr instruction:

```
mov w, >>fr
```

Here, the “>>” has a different meaning because this operation is performed at run-time: The contents of fr is rotated right by one bit (with carry), and the result is then copied into the w register without changing the original value in fr.

The >< Operator

Q: I have been trying to use the Reverse Bits Binary Operator as listed on page 71 of the SX-Key Blitz Manual 2.0. The symbol for this operation is >< but I can find no examples of its use. What is it good for, and how does it work?

A: The >< operator (similar to the >> operator) can be used in expressions to be evaluated by SASM while assembling a program, i.e. it is not executed by the SX at run-time.

Actually, the >< operator is a bit tricky, and will be rarely used. Here is an example:

```
foo1 = %10111000
foo2 = foo1 >< 6
```

This assigns a value of %10000111 to foo2 – why?

Well, >< 6 means: “Take the lower 6 bits of the value, but them in reverse order, and replace them in the original value”.

Thus, foo2 is composed of

`10` (bits 7 and 6 remaining unchanged) and

bits 5…0 which originally were

`111000`, now placed in reverse order, yielding in `000111`, so the final result is

`10000111`.

As mentioned before, the >< operator is evaluated at assembly-time, and not by the SX when it executes a program. Nevertheless, you should keep in mind that some SX instructions also allow to modify a value when it is copied from one register into another. For example,

```
mow w, /Temp
```

copies the inverted value contained in Temp to w, similar to

```
not Temp
mov w, Temp
```

but without changing the contents of Temp, and

```
swap Temp
```

exchanges the upper four and the lower four bits in Temp.

Breakpoints (N2)

Q: When should I use the BREAK directive, and how shall I use breakpoints with the SX-Key debugger?

A: A breakpoint is used while debugging an SX Assembly program in RUN mode (i.e. at full speed) to stop program execution at a pre-defined instruction in the code. After program execution has stopped, you may inspect the contents of certain registers, continue program execution in single steps, in Walk mode, or re-enter the Run mode by clicking the corresponding buttons in the Commands window.

To set a breakpoint while the debugger is active, scroll the debugger's list file window until the line where you want to set the breakpoint becomes visible. Left-click on that line to activate the breakpoint. The line will then be displayed with a red background indicating that the breakpoint is set there.

When you run the program at full speed (click the "Run" button to start), program execution will stop after the instruction with the breakpoint has been executed, and the debugger returns to single-step mode again. Note that the program will stop only after the "breakpointed" instruction has been executed. This sometimes is a bit annoying. For example, when you set a breakpoint on a line containing a JMP or CALL instruction, the program will halt on the first instruction of the jump target, or the first instruction of the sub-routine. In order to single-step the JMP or CALL, place the breakpoint on the instruction immediately preceding the JMP or CALL instruction.

You can set a new breakpoint at any time – just left-click on another instruction line to cancel the former breakpoint (if any), and to activate the new one.

Please note that – due to the SX-internal structure – it is only possible to set just one breakpoint at any time. When you define a new one, the former breakpoint will always be cleared automatically.

Also – due to the SX-internal structure – it is not possible to set a breakpoint on an instruction line containing a NOP. If you do this, the breakpoint will be ignored, and program execution continues at full speed.

To clear a breakpoint without setting a new one, simply left-click on the line with the current breakpoint to toggle it off.

In order to set a "general breakpoint", i.e. one you might want to be automatically set whenever you re-enter the debugging session of a program, you may insert a BREAK directive in the source code immediately before the instruction line where you want the breakpoint. When the debugger loads the file, it automatically activates a breakpoint on the next code line following the BREAK directive. While the debugger is active, you may select another breakpoint at any time, as described before, and the "general breakpoint" will be removed.

When you decide to define another "general breakpoint", just delete the BREAK directive in the source code, and place it at another location. As long as you did not make any other changes to the source code, it is not necessary to re-assemble and to transfer the program code to the SX again. Therefore, you may use the "Run – Debug (reenter)" menu option to restart the debugging session.

Please note that a source code file may only contain one BREAK directive at a time (because the SX device allows for one active breakpoint only). When your code contains more than one BREAK directive, the Assembler will report an error.

Please also note that the "Poll" option is not available during "Run" mode as long as a breakpoint is active.

Chip connection failed Error

Q: When I try to program an SX device with the SX-Key, the IDE reports a “Chip Connection Failed Error”, what is the reason for this?

A: There are several possible reasons for this error message:

- Sometimes, this message randomly comes up due to timing issues, or spikes. Before checking the next points, just confirm the message, and try again. When you get that error continuously, check the next items:

- Some external component, like a resonator, crystal or clock generator may be connected to the OSC1 and/or OSC2 pins, causing too much load for the SX-Key – remove these components, or pull the jumper (when available) connecting these components to the OSC1 pin.

- The supply voltage is not stable – must be close to 5V during programming. Although the SX chips can operate in a wider range of supply voltages, the SX-Key can’t. If you power your system from an adjustable supply, try to slightly increase or decrease the voltage around 5V.

- When using a prototype board, like the SX Tech board with a voltage regulator for the supply voltage, make sure that the regulator’s output filter capacitor is at least 22 µF. If not, solder one in parallel to the existing capacitor, or plug one into the available Vdd and Vss connectors of the board. Make sure that the polarity of this capacitor is correct, the positive lead (usually, the longer one) goes to Vdd, and the negative lead (usually the shorter one, marked with a white band on the can) goes to Vss. **CAUTION: Wrong polarity might cause the capacitor to explode!**

- The MCLR pin must be held high during programming. Make sure that it is connected to Vdd (eventually with an external pull-up resistor).

- The SX-Key is plugged in the wrong way – make sure that its orientation is correct.

- The header pins for the SX-Key, or the 4-pin connector of the SX-Key may be contaminated, or the SX-Key connector may be worn out. Unplug, and plug in the SX-Key a couple of times, or use some cleaning solution to remove any contamination. If necessary, replace the 4-pin connector on the SX-Key.

- The SX device is inserted the wrong way into its socket – make sure that the chip’s index mark matches the mark on the socket.

- The SX device is damaged – if you have another one on hand, try if this programs ok.

- No SX device is inserted at all.

Clock Jitter caused by the SX-Key

Q: In doing some critical timing debugging at 50 MHz using the SX-Key, I noticed up to 200 nanosecond jitter in my interrupt routine when there ought to be none. Reprogramming the device in non-debug mode, and plugging in the 50 MHz resonator made it rock-stable. What's the problem here?

A: The SX-Key device uses a PLL chip to generate the various clock frequencies. It is typical for such PLL devices that they cannot precisely generate all the frequencies in the specified range. Either, the resulting clock signal is a bit off the requested value, or it has some jitter to achieve an average frequency that matches the requested one.

For such time-critical applications, you are better off to replace the SX-Key with the final clocking device when it comes to test the timing of the application.

The Debugger starts in running mode

Q: On my SX 28, when I choose Debug, the debug window comes up RUNNING and the options for Polling, Run, Stop etc. are gray and I cannot single-step through my program.

A: One reason might be that you have previously programmed the SX selecting “Run – Program” and then selected “Run – Debug (reenter)”.

The reason is that there is a difference between “Program” and “Debug”. When you choose “Program”, the only code sent to the SX is the code you wrote. When you select “Debug” instead, not only is your code sent, but a little piece of special code is also appended. This additional code is required to enable the SX-Key to properly control and communicate with the chip while debugging.

The option “Run – Debug (reenter)” is meant to continue a previous debugging session, assuming that no changes have been made to the code in the meantime. Therefore, the code is not sent into the SX device again, saving the time required to do this. The IDE expects that the required debugging code is available in the chip, though. When this is not the case (because the chip was not programmed for debugging), the debugger can’t communicate with, or control the chip. Therefore, it indicates “RUNNING” only, and does not allow any other options.

Another reason for the debugger starting in running mode is that a resonator, crystal, or an external clock source is connected to the OSC1 and OSC2 pins in parallel to the SX-Key. Although the chip can usually be programmed in this configuration, debugging is not possible. Remove or disconnect the external clocking device to allow for debugging.

Another cause for the debugger to display the “RUNNING” status is when the MCLR* pin is pulled low by some reason. In this case, the SX is definitely not running but held in reset state. It seems as if the Debugger reports “RUNNING” whenever it cannot communicate with the SX.

The DW Directive

Q: Where do I place a DW directive in my source code, and how do I retrieve data defined by this directive?

A: The DW directive can only be placed in the code section of the source code, i.e. after an ORG directive referencing program memory as DW defines data words in program memory.

As locations in program memory are 12 bits wide, you can use DW to define values between $000 and $FFF. In order to retrieve data defined by DW directives, use the IREAD instruction. Here is an example:

```
      ; …
      mov m, #Text >> 8
      mov w, #Text
      iread
      ; …

      ORG $200
Text  DW 'Hello'
      DW 13, 10
      DW 0
```

As you can see, parameters for the DW directive can be single values, a list of two or more numerical values separated by commas, but also character strings within single quotes. When the parameter is a list of values, or a character string, the assembler will set aside consecutive locations in memory for each of the values or characters in the string.

The IREAD instruction (Indirect READ) is used to read a word from program memory. It expects the upper four bit of the read address in the lower four bits of the m register, and the lower eight bits in the w register. After execution of an IREAD, the result is in m:w with the upper four bits of the result in the lower four bits of the m register, and the lower eight bits in the w register.

In the example, above, all values specified with DW directives are below 256, i.e. after an IREAD, m is always zero and can be ignored.

The code, shown above, will read the first character “H”, and place its ASCII equivalent ($48) into the w register, with the m register containing zero. In order to read a table like in the example, you usually would use a loop construct with an incrementing pointer, copied into w before executing the next IREAD instruction until it returns 0 in the w register, indicating the end of the string.

The END Directive

Q: What is the END directive good for in an assembly program – does it terminate a program, like in BASIC?

A: The END directive does *not* terminate an SX program – it simply "tells" SASM, that no more valid statements to be handled by the assembler will follow, i.e. any text after the END statement will be ignored by SASM. This is handy when you want to add some information for documentation purposes at the end of a source code file, like in this example:

```
Main
      ;
      ; many lines of code
      ;
      mov   ra, #$0F
Forever
      jmp   Forever
END

History:

First version:    Finished by the end of the ice age
Current version:  Modified yesterday, i.e. I don't remember when.
```

Please note that the last line assembled by SASM is the `jmp Forever` instruction which brings the program execution into an endless loop, and the text beginning with `History:` is ignored. If this last `jmp Forever` instruction were missing, the SX would continue reading subsequent locations of the program memory (where each word usually contains the instruction code $FFF for NOT W), execute these instruction codes, until the program counter reaches the topmost location, where a jump instruction to the main entry point is found and executed. In other words, the whole program would be repeated in an endless loop, which is usually not what you want.

Instead of an endless loop, you can use a SLEEP instruction instead, to keep the SX from running into unprogrammed areas of the program memory, like in

```
Main
      ;
      ; many lines of code
      ;
      mov   ra, #$0F
      sleep
```

The FREQ Directive – SX Clocking

Q: I've managed to get my program running fine using the SX-Tech board, but when I program the chip on my prototype board, the SX28 doesn't seem to be running. I've checked the pin diagrams and it seems to be wired correctly. Am I supposed to be using an external oscillator?

A: There are some important points to be considered when switching between a “debugged” SX configuration to a “stand-alone” version.

When the SX is programmed for debugging, some additional code to support the communication with the SX-Key is added to your application-specific code. This additional code will be transferred to the SX when you select the “Debug” option from the SX-Key’s “Run” menu option.

When you select the “Program”, or “Run” options from this menu instead, no debugging code will be transferred.

While Debugging:

The SX-Key generates the clock signal for the SX under test where the clock frequency is selected according to the FREQ directive found in the source code file. In order to successfully run the debugger, there must not be no DEVICE WATCHDOG directive active in the program. In addition, when you program the SX for debugging, some additional code besides your application-specific code is transferred into the SX program memory to support the communication with the SX-Key while debugging an application. This requires some free program memory for the additional code, and SASM will report an error when your application makes use of it.

While Running:

When you select the “Run - Run” option from the SX-Key IDE, the SX device will be re-programmed without the additional code required for debugging, i.e. for “stand-alone” mode, but the SX-Key will still supply the clock signal with the frequency specified by the FREQ directive in your code.

When the Clock Option is selected:

When you select “Run – Clock” option from the SX-Key IDE, it is assumed that the SX under test has been programmed for “stand-alone” mode before (i.e. without the additional debug code). The SX-Key supplies the clock frequency where you can select the desired frequency from the dialog box that opens when you choose “Run – Clock”, i.e. the FREQ directive does not control the clock frequency anymore.

“Stand-Alone” Mode:

Before removing the SX-Key to run the device “stand-alone”, make sure that the SX is programmed without the additional debug code. If necessary, Select “Run – Program” to re-program the SX (If you did a “Run- Run” before, the SX is already programmed for stand-alone mode, so there is no need to execute “Run – Program” again. In this mode, the clock is either generated by the internal RC oscillator, the internal clock generator driving an external RC network, a resonator, or a crystal, or by an external clock signal fed into the OSC1 pin. The clock frequency is determined by the components used to generate the clock, and no longer by the SX-Key.

Please note that the FREQ directive has no influence on the clock frequency in “stand-alone” mode. Also make sure that the code containing a DEVICE OSCxx directive does match the clock device you are using.

INDF

Q: What is the meaning of INDF?

A: INDF is a synonym for IND. Both represent RAM address 0, and are used to indirectly address a RAM location, like in

```
mov FSR, #$08     ; FSR "points" to $08
mov w, Temp
mov IND, w        ; w is copied into the register
                  ; whose address is contained
                  ; in FSR ($08 in this example)
mov INDF, w       ; Same as the previous operation
```

The IRC_CAL Directive (N2)

Q: What is the IRC_CAL directive good for, and when do I need it? Is it OK to ignore the “No IRC_CAL directive” warning?

A: The internal RC network for generating the clock has certain production tolerances. Therefore, three bits in the bits in the FUSEX configuration register allow to adjust the frequency close to 4 MHz.

When your program contains the IRC_CAL IRC_4MHZ directive, the SK-Key software automatically performs a calibration process, i.e. it measures the clock frequency generated by the SX-internal RC network, and adjusts the FUSEX bits until the frequency is as close to 4 MHz as possible.

When you don’t use the internal RC network for clock generation, you should skip this calibration process as it requires some seconds, each time the device is programmed. Either use IRC_CAL IRC_SLOW or IRC_CAL IRC_FAST to instruct the SX-Key IDE to set the three FUSEX bits to the minimum (000), or maximum (111) possible frequency.

Please note that even with a calibrated internal RC network the resulting clock frequency is not too stable. Changes in supply voltage, and ambient temperature have influence on the actual frequency. Therefore, you should use the internal RC network for clock generation only, when the application does not require an exact timing. The same is true to a similar extend for an external RC network. When timing is important, use a crystal, resonator, or an external stable clock source.

List Files not saved in the source directory

Q: For SXSim, I need the list files but SASM does not seem to create them – what's wrong?

A: Actually, nothing is wrong – it is just a matter of configuration.

In order not to fill your hard disk with a bunch of LST, OBJ, SYM, and ERR files which are usually not required, SASM, by default, writes all these files created during an assembly session in the "SASM Output Directory". After a successful assembly, SASM automatically deletes all these files.

If you want the files to "stay alive", select the "Run – Configure – Assembler" menu option in the IDE, and un-check the "SASM files to 'SASM Output' dir" option. This instructs the IDE to save these files in the folders where the associated source files are located, and not to delete them after a successful assembly.

When you are using the SX/B compiler, select "Run – Configure –SX/B Compiler" and uncheck the "SX/B files to 'Output Files' dir" in order to send the temporary files into the source file directories, and leave them un-touched after a successful compile.

Literal truncated to 8 bits warning

Q: Is there something wrong with my code when I get a “Literal truncated to 8 bits” warning?

A: In almost all cases, the answer is: No.

A very common situation for such a warning is when you initialize the w register with a negative value before executing a RETIW instruction at the end of an interrupt service routine, like in

```
mov w, #-163
retiw
```

SASM converts the decimal literal -163 into its binary equivalent which is internally represented as 64 bit value: 1111 1111 1111 1111 1111 1111 0101 1101, or FFFFFF5D in hex (the 2-s complement representation). As the w register is just eight bits wide, only the lower eight bits are copied into w (which is exactly what’s required here), and the higher bits are truncated.

Another situation is when you initialize w as part of an m:w address for an IREAD instruction, like in

```
mov m, #TextTable >> 8
mov w, #Hello
```

Here, #Hello represents the 12-bit address of some location in program memory. Again, w can only store the lower eight bits of that address (the upper four bits are truncated), and this is why the warning will be generated.

In order to suppress this warning, place a

```
LIST Q = 37
```

directive at the very beginning of your source code.

The NOP Instruction

Q: When I run the debugger for this blinking LED code, it steps the first NOP, and then skips to the next instruction which is SETB. Is this normal (it seems like it only executes one NOP, and ignores the other two)?

```
Loop:
      clrb  LED
      nop
      nop
      nop
      setb  LED
      jmp Loop:
```

A: Yes, this is normal. The SX chip does not allow you to set a breakpoint on NOPs, or single-step them. This means, even though they are executed, all debuggers will skip right through them during debugging.

It is also typical for any SX debugger (due to the SX internals) that when a breakpoint is active, the "breakpointed" instruction will be executed before the program halts. This sometimes can cause confusion, when you set the breakpoint on a JMP instruction, for example. The program will stop at the first instruction at the jump target, and not on the JMP instruction.

Sometimes, while debugging, it is helpful to insert an extra instruction that does not change any register contents, like

```
      and w, #%11111111
```

as an "anchor" for a breakpoint. Note that this instruction will set the Z flag in case w has a value of zero, and clear it if this is not the case. If this is a problem, use a pair of SWAP instructions, like

```
      swap $08
      swap $08
```

instead. The SWAP instruction (except NOP) is the only one that does not affect any flag, and two subsequent SWAPs on the same register leave its value unchanged as well.

The ORG Directive

Q: I'm having some trouble with a few key concepts, especially memory mapping. I understand that the lower half of each bank is always the same set of registers ($00-$0F), and that $03, $05, etc. represent different banks of data registers (8 total) and that there are 4 program data pages, but I get confused with using the ORG instruction.

A: Basically, the ORG (Origin) directive tells the assembler at which starting address it shall continue either setting aside space in data memory, or generate code in program memory. For both sections, the ORG directive is used, and the assembler is “clever” enough, to differentiate between data and code memory.

Here is a code snippet as an example for the data memory:

```
; Global variables
      org 8
Global_ByteCount ds 1
Global_Index     ds 1
; more (max. 8 variables)

; Serial variable area
      org $10
Serial = $
Serial_TxHighPC ds 1
Serial_TxLowPC  ds 1
; more (max. 16 variables)...

; FIFO buffer
      org $30
Buffer          ds 16   ; 16-byte array, will
                        ; be indirectly addressed

; Miscellaneous stuff
      org $50
Misc = $
Misc_FIFOHead   ds 1
Misc_FIFOTail   ds 1
; more (max. 16 variables)...

      org $70
Free1 = $

      org $90
Free2 = $

      org $B0
Free3 = $

      org $D0
Free4 = $

      org $F0
Free5 = $
```

Here, the ORG directives tell the assembler where (i.e. in which bank) it shall set aside memory for the variables. It is a good idea to always also define a symbolic name for each bank, like with `Serial = $`. Later, when you want to access a variable, be careful that the most recently executed BANK instruction

has selected the bank, where the variable is located in. If necessary, insert an instruction like "bank Serial" before accessing TXHighPC in the sample code, for example.

It is also a good idea to strictly use the naming convention <Bank>_<VarName>, i.e. the first part of each name consists of the bank name. This makes the code easier to read because you always know, where a variable is located, and helps you to check if the right bank is selected.

Each memory bank has a size of 32 bytes where the lower half, i.e. the first 16 bytes are always mapped to bank 0, no matter which bank is actually selected. Within the first 16 lower bytes of bank 0, only the higher eight (six on SX 48/52 devices) locations are free to be used as variable space because the lower eight (or ten on SX 48/52 devices) bytes are used as special registers, e.g. the PC, the FSR, the ports, etc.

Therefore, there is the ORG 8 in the above example. As these upper eight bytes in bank 0 can always be accessed, no matter which bank is actually selected, they are usually called "Global Variables".

Besides this, as mentioned before, the ORG directive is also used to specify the starting points of program code, like in

```
      org 0

; The interrupt service routine must always start at $000
; (if any)
ISR
; some code
      mov w, #IntPeriod
      retiw

      org $100

; The Main program starts at $100
Main
; more instructions following…
```

As your program grows, and the code gets longer, it may happen that it no longer fits into one page of program memory. When the assembler generates the instruction code, it fills consecutive locations in program memory when it does not come across an ORG directive. This also means that the assembler does not check if the generated code crosses code page boundaries. Later, when the SX executes this code, this is also not a problem as long as the execution is “linear”, i.e. has no branches or as long as jumps are executed into the page from where the code originated. Although it can cause a nightmare when JMPs refer to targets in the next code pages (unless you insert a matching PAGE instruction before, or use the @ modifier with JMP/CALL instructions).

Nevertheless, the better solution is to use a “trick” for detecting when code across page boundaries is generated. Here is an example:

```
      org $200
      nop

      org $400
      nop

      ; Place more nop instructions at the beginning of all other pages
```

Now, when the code generated by the assembler crosses a page boundary, the assembler detects that there already is an instruction (the NOP), and generates an error message that this location is not empty.

The DEVICE OSCxx Settings

Q: The SASM assembler allows for various DEVICE OSCxx settings – what are thy good for, and which one should I use?

A: The DEVICE OSCxx directive determines the setting of the three FOSC2 to FOSC0 bits in the FUSE device configuration register when the SX is being programmed. These bits control the gain of the SX-internal oscillator driver in order to addapt it to external components like crystals, resonators, or RC networks, depending on the clock frequency generated by those devices. The table, below, lists the available DEVICE OSCxx directives, and the resulting FOSC bits and their meanings:

DEVICE	FOSC2..0	Meaning
OSCLP1	000	low power crystal (32 kHz)
OSCLP2	001	low power crystal/resonator (32 kHz to 1 MHz)
OSCXT1	010	normal crystal/resonator (32 to 1 MHz)
OSCXT2	011	normal crystal/resonator (1 MHz to 24 MHz)
OSCHS1	100	high speed crystal/resonator (1 MHz to 50 MHz)
OSCHS2	101	high speed crystal/resonator (1 MHz to 50 MHz)
OSCHS3	110	high speed crystal/resonator (1 MHz to 75 MHz)

As you can see, for some frequencies, there are several options you might select. In order to minimize RFI generated by the SX, you should select the lowest gain that still safely starts the clock oscillator at power-on.

Depending on the clocking device you are using (crystal or resonator), it may be necessary to connect an external resistor between the OSC1 and OSC2 pins of the SX, and capacitors from ground to the OSC1 and OSC2 pins. Please refer to the “Oscillator Circuits” sections in the SX data sheets for more details.

The SX also has an internal crystal/resonator feedback resistor (1 MΩ) that may be enabled or disabled by setting or clearing the IFBD bit in the FUSE register.

By default, SASM assumes that this resistor shall be enabled, i.e. it generates code to set the IFBD bit while programming the SX device. Use the DEVICE IFBD directive to have this bit cleared, and the internal feedback resistor disabled. In this case, you should provide an external resistor.

Using RC Networks for Clock Generation

The DEVICE OSCRC directive sets all FOSC bits in the FUSE register which configures the SX-internal clock driver for an external RC network with a resistor connected between Vdd and the OSC1 pin and a capacitor connected between ground (Vss) and the OSC1 pin. You should consider using an RC network only when the precision of the generated clock frequency is not an issue as the frequency is a function of the supply voltage, temperature, and component tolerances.

As another option, you may also use the SX-internal RC network for clock generation. Use one of the DEVICE OSC4MHZ, OSC1MHZ, OSC128KHZ, or OSC32KHZ directives to activate the internal RC network (i.e. FUSE bit IRC will be cleared, and bits DIV1 and DIV0 will be configured according to the selected clock frequency).

As with an external RC network, the internal RC network does not guarantee for a stable system clock. You may add the IRC_CAL IRC_4MHZ directive in you source code which instructs the SX-Key to perform a calibration of the internal RC clock when the SX is programmed by adjusting the setting of bits IRCTRIM2…0 in the FUSEX device configuration register. Nevertheless, the clock frequency will still remain a function of the supply voltage, ambient temperature, and other factors – so you should only use the internal RC clock when precise timing is not an issue.

The SX-Key IDE “Run” Menu Options (N2)

Q: What are the various options in the „Run“ good for?

A: Most options in the “Run” Menu that are described, below, are used to program an SX device, and to execute a program on the SX device. There are two major modes for programming the SX: “Stand-Alone”, and “Debug”.

When programmed for “Stand-Alone” mode, the SX program memory just contains the machine code generated by the SASM assembler according to the translated source code. In Stand-Alone mode, the SX either is clocked by its internal RC network, an external RC network connected to the OSC1 pin, a resonator or crystal connected to the OSC1 and OSC2 pins, or an external clock source connected to the OSC1 pin.

When programmed for “Debug”, the machine code transferred into the SX program memory contains additional code that is required to allow the SX-Key to communicate with the SX device in order to debug the program. In this mode, the SX device is always clocked by the SX-Key.

It is important to keep in mind, that an SX programmed for “Stand-Alone” mode cannot be debugged. When you try to do this, the debugger will come up with the status “Running”, not allowing for any interaction with the SX. On the other hand, a device programmed for “Debug” can’t run stand-alone. When you try this, i.e. connect a clock source to the device, it will not start program execution.

Assemble
This option launches the SASM assembler, in order to translate the source code currently displayed in the editor window. When there are no errors in the source code, the message “Assembly successful” will be displayed. Should there be errors in the source code, they will be reported. You will mostly use the “Assemble” option to check the source code you are working on for errors. Note that this option does not transfer code into an SX device. Therefore, it is not necessary to have the SX-Key connected or powered.

Program
This option first performs the same steps as the “Assemble” option. When the source code could be successfully translated, the generated machine code will be automatically transferred into the SX program memory for “Stand-Alone” operation. Therefore, it is important that the SX-Key is connected to a COM port of the PC on one end, and to the four-pin programming header of the SX device on the other end. It is also necessary that both, the SX and the SX-Key are powered with 5 Volts.

While programming, a dialog box with a progress bar is displayed. When this box is closed, programming is complete. You may then power-off the SX, disconnect the SX-Key and connect one of the clocking devices, as described before. When the SX is configured to use its internal RC network, no external devices are required.

Run
This option first performs the same operations as the “Program” option. After a successful assembly and after the program for “Stand-Alone” mode has been transferred into the SX program memory, the SK-Key is activated as external clock source for the SX device. In other words, the SX-Key must remain connected to the SX device, and the SX-Key and the device must remain powered.

The clock frequency generated by the SX-Key is determined by the FREQ directive in the source code. When there is not such a directive, 50 MHz is assumed by default.

Debug
This option performs the same steps as the “Program” option with the exception that the code sent into the SX program memory contains additional code required for debugging. After the program transfer is completed, the debugger is automatically launched in Idle mode, ready for debugging the program. When you use the debugger’s “Run” mode, the SX-Key will generate a clock frequency according to the FREQ directive in the source code (or 50 MHz by default).

Debug (reenter)
This option simply launches the debugger without transferring program data into the SX program memory before. *Therefore, it is very important that the code contained in the SX program memory matches the program code held in the SX-Key IDE.* Simply spoken, you may use the “Debug (reenter)” option only when you did not make any changes to the source code since you have used the “Debug” option last.

There are two exceptions though: You may change the parameter of the FRQU directive to let the SX-Key generate another clock frequency, or you may add a BREAK directive, or change the position of an existing BREAK directive without the need to re-assemble the source code using the “Debug” option.

View List
This option invokes SASM, like the “Assemble” option does. After a successful assembly, the SX-Key IDE opens a new window showing the contents of the list file generated by SASM. This list file gives an overview about the code generated as the result of the source code lines plus other information. At the end of the list file, you will find a “Cross Reference” section showing all the symbols used in your program.

Launch SXSim
This option is new to version 3.1 of the SX-Key IDE. It first invokes SASM, like the “Assemble” option does. After a successful assembly, it launches SXSim with the current list file name as parameter, i.e. you can immediately start simulating the current program in SXSim.

Clock…
Similar to the “Run” option, this option activates the SX-Key as external clock generator for the SX device but without re-assembling the program, and transferring it into the SX program memory. Therefore, make sure that the SX has been programmed for stand-alone mode (either by “Assemble”, or by a previous “Run”). To use the clock, the SX-Key and the SX device must be connected and powered as described before. When you select this option, a dialog box opens, allowing you to specify the clock frequency to be generated by the SX-Key. Please take care not to “overclock” the SX. Although the clock dialog allows for higher frequencies, you should not go beyond the maximum of 50 or 75 MHz, depending on the SX device you are using. Click the “Ok” button to accept the selected frequency and to close the dialog. After closing the dialog, it is no longer necessary to keep the SX-Key connected to the PC’s COM port. The SX-Key will clock the SX device as long as power remains on.

Using SX/B
The options described before, are also valid when you use SX/B programs. In addition, the SX-Key IDE automatically launches the SX/B compiler in order to translate the SX/B source code into an assembly source code before launching SASM to generate the machine code. In case, the SX/B compiler reports errors, they are reported, and SASM will not be launched.

The RTCC Input

Q: I'm trying to determine whether the RTCC pin should be tied to Vss or Vdd when it is not used for external triggering. What is the best solution?

A: When the RTCC input is not used, it is a good idea to tie it to some defined level instead of leaving it “floating”. A good idea might be to connect it to Vdd via a pull-up resistor of say, 4.7 kOhm.

Imagine what would happen in an "ISR-less" application when you have set bit 5 in the OPTION register by some reason (i.e. RTCC roll-over interrupt enabled), with a floating RTCC input randomly incrementing the RTCC, and finally causing a jump to $000 on a roll over. As your program does not contain any specific ISR code, it might restart, or perform some other crazy operations.

Supply Voltage for SX Devices

Q: What is the allowed range of the supply voltage for SX devices?

A: The SX datasheets specify 2.7 to 5.5 V as typical supply voltages, and an absolute maximum rating of 7.0 V on the Vdd pin with respect to the Vss pin.

When powering the SX with voltages below 5 V, make sure that other external components connected to the SX are also specified for such lower voltages, and also make sure that the SX brownout detection is either turned off (DEVICE BOROFF), or set to an appropriate value (4.2, 2.6, or 2.2 V).

Please note that the SX-Key requires a 5 V supply for proper operation. This means you will have to power the SX system under test with 5 V when it also powers the SX-Key.

SX-Key not found on COMx Error

Q: When I try to program an SX device with the SX-Key, the IDE reports an “SX-Key not found” Error”, what is the reason for this?

A: There are several possible reasons for this error message:

- The SX-Key is not connected to the PC’s COM-port specified in the configuration section of the SX-Key IDE.
- The SX device and/or the SX-Key are not powered.
- The supply voltage is not stable – must be close to 5V during programming. Although the SX chips can operate in a wider range of supply voltages, the SX-Key can’t. If you power your system from an adjustable supply, try to slightly increase or decrease the voltage around 5V.
- The SX-Key is plugged in the wrong way – make sure that its orientation is correct.
- The SX device is inserted the wrong way into its socket – make sure that the chip’s index notch matches the mark on the socket.
- Some versions of Windows cause problems when the internal FIFO buffer for the port that’s attached to the SX-Key is configured too large. Check the FIFO buffer size, and try to reduce it to 3 to 4 bytes, or even completely turn off the FIFO buffer.
- The SX device or the SX-Key is damaged – if you have another one on hand, try if this programs ok.

TAB Settings in the Editor

Q: I commonly port an assembly program from one computer to another. It seems that this often leads to losing the formatting of comments. Opening a program that was previously on a different computer, with the comments lined up all nice and pretty (starting in same column), some of the comments will no longer be lined up.

A: In order to keep the same formatting on different machines, make sure that you have specified the same Tab size settings. Select “Run – Configure” from the main menu, and then “Editor” from the Category list, and make sure that the “Tab size” setting is equal on both machines.

Top Files

Q: The SK-Key IDE allows to set a file as “Top File” – what is this good for?

A: Top Files are quite handy when you have a project consisting of more than just one source code file. You may, for example, place standard code used to define the DEVICE settings in a separate file, or place the code used to clear up the RAM at startup in second file, and frequently used macro definitions in another file.

In your “Main” file, you would then use INCLUDE directives to have the assembler insert the contents of theses additional files at the right places. This works fine as long as the “Main” file is visible in the editor window, when you select any option from the “Run” menu that causes a new assembly of the file. On the other hand, when the editor window displays one of the include files while you start an assembly, the assembler would translate just the contents of this file which most likely would end in a number of warnings or errors, and the resulting code (if any) would not contain your full project code.

This is when “Top Files” come to play a role:

Select your “Main” source code file, i.e. the file containing the INCLUDE directives for all the other files used by the project from the file list in the window, left of the edit window by left-clicking on its name. Next do a right-click on the name, and select “Set as Top File” from the context menu that drops down. After having done this, you will notice that the file name is shown in bold, indicating that this is a “Top File” now.

After you have defined a “Top File”, any of the associated include files may be visible in the editor window when you launch the assembler – it will always start assembly by reading in the “Top File” first which is necessary for a complete assembly of the project.

The SX-IDE manages an internal list of up to 20 “Top Files”, i.e. whenever you open a source code file that has been marked as a “Top File” before, it will receive this attribute again when re-opened.

You can remove a “Top File” from that list at any time by selecting and then right-clicking on the “Top File’s” name. In the context menu, select “Remove from known Top File list”. This also removes the “Top File” attribute from the selected file. You can also right-click on the name of a selected “Top File”, and click on “Set as normal file” to remove the “Top File” attribute in order to define another file in the project as the “Top File”.

Unused I/O Pins

Q: What shall I do with unused I/O pins – can I leave them open?

A: In general, nothing would get damaged when you leave unused pins open but when configured as inputs, they have high impedance, so they might “float”. Therefore, it is a better solution to assign a defined status to unused pins:

- Configure the pins as inputs, and either activate the internal weak pull-up resistors, or connect external resistors (say 4.7 kOhm) between each pin and Vdd (pull-up), or Vss (pull-down). When using external resistors, do not activate the internal weak pull-up resistors.

- Configure the pins as outputs, and drive the pins to either low, or high level.

Please note that two other input pins are available: The RTCC pin (add an external pull-up, or pull-down resistor), and the MCLR* pin which requires special attention. If this is left floating, the SX could randomly perform resets. Therefore, it is important to pull this pin up to Vdd using a resistor (4.7 to 10 kOhm would be fine).