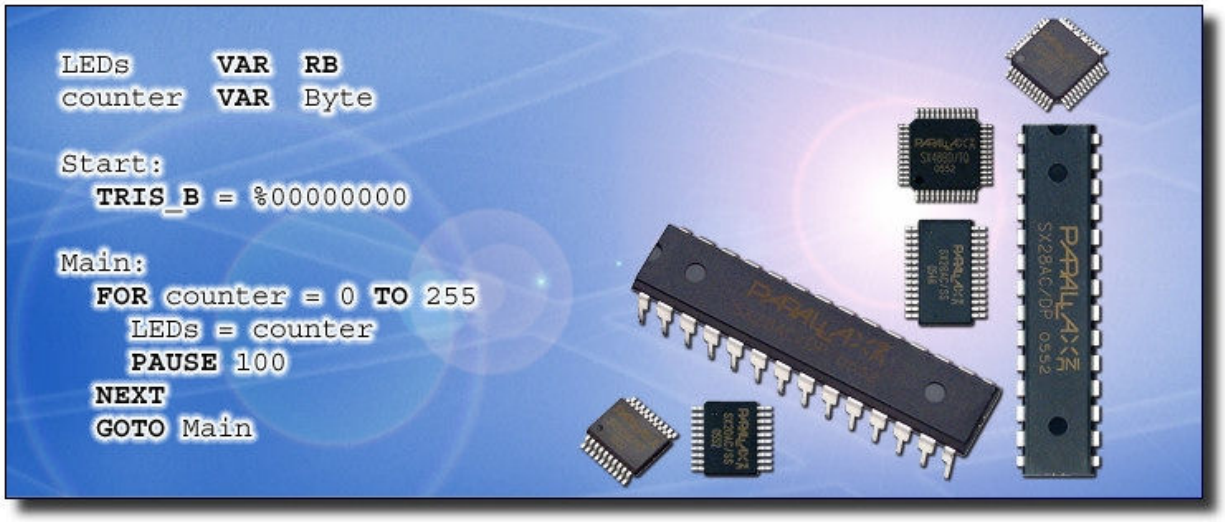
SX/B Online Help



599 Menlo Drive, Suite 100
Rocklin, CA 95765

Tel: 916-624-8333
Fax: 916-624-8003

Toll-free Sales: 888-512-1024 (Continental US only)
Toll-free Support: 888-99-STAMP (Continental US only)

Visit us on the web:www.parallax.com

SX Microcontrollers Forum: forums.parallax.com/forums

Comments and feedback: editor@parallax.com

Copyright © 2004-2007 Parallax, Inc.
All Rights Reserved

version 1.51.03 -- 14 MAY 2007

SX-Key and BASIC Stamp are registered trademarks of Parallax Inc. SX/B and the Parallax logo are trademarks of Parallax Inc. SX is a trademark of Ubicom; used with permission. 1-Wire is a registered trademark of Maxim/Dallas Semiconductor. I²C is a registered trademark of Philips Corporation. Other trademarks herein are the property of their respective holders.

SX/B Compiler Overview

The SX/B compiler is a BASIC language compiler for the Parallax SX family (SX20, SX28, SX48) microcontroller (as well as the older Ubicom™ SX18 and SX52), and was designed to meet two specific goals:

1. Expedite the task of the professional engineer by creating a simple, yet robust high-level language for the SX microcontroller. This allows SX-based projects to be prototyped and coded quickly.
2. Assist the student programmer wishing to make the transition from pure high-level programming (i.e., PBASIC) to low-level programming (SX assembly language).

SX/B is an non-optimizing, inline compiler. What this means is that each BASIC language statement is converted to a block of assembly code in-line at the program location; no attempt is made to remove redundant instructions that would optimize code space. This allows the advanced programmer to modify code as required for specific projects, and -- perhaps more importantly -- provides an opportunity for the student to learn SX assembly language techniques by viewing a 1-for-1 (from BASIC to assembly language) output.

Conventions Used in this Document

In syntax descriptions, curly braces {} are used to indicate *optional* items. For example:

PULSOUT *Pin, Duration {, Resolution}* In this case, the parameter for *Resolution* is optional.

In syntax descriptions, brackets [] indicate that the parameter must be one of the presented items (separated with the pipe | character). For example:

DEVICE *[SX18 | SX20 | SX28 | SX48 | SX52] {, ...}*

In this case, the **DEVICE** directive must indicate SX18, SX20, SX28, SX48, *or* SX52 (and may also include other *optional* items).

Example code is presented on a blue-tinted background:

```
Main:
  FOR RB = 0 TO 255
   PAUSEUS 10_000
  NEXT
  END
```

What's New in SX/B?

Version 1.51.03 -- Release date 05 JUL 2006

- **SUB**s may have 5 parameter bytes ONLY if it requires exactly 5

```
mySub SUB 5 ' This is legal
mySub SUB 2,5 ' This is NOT legal
```

- Added and enhanced SX/B commands:
  - **LOOP NEVER** option
  - **SUB**...**ENDSUB** and **FUNC**...**ENDFUNC** structure
  - **ANALOGIN** command
  - **READINC** command
  - **NOP** command
- Added Conditional Compilation Directives:
  - '{$**DEFINE** *name*}
  - '{$**UNDEFINE** *name*}
  - '{$**INFDEF** *name*}
  - '{$**INFDEF** *name*}
  - '{$**ELSE**}
  - '{$**ENDIF**}
  - Predefined "SX18", "SX28", "SX48", or "SX52" by DEVICE directive
  - Directive may NOT be nested.

```
' Example of Conditional Compiling Directives

  '{$DEFINE USE_RB} ' Remove this line to use RC

  '{$IFDEF USE_RB}
  LEDS PIN RB
  '{$ELSE}
  LEDS PIN RC
  '{$ENDIF}
```

Version 1.50.01 -- Release date 05 JUL 2006

- Added Word (16-bit) variables
- 16-bit pseudo-ports: RBC (SX28/48/52), RCD (SX48/52), RDE (SX48/52)
- Added defined functions (**FUNC**) to return up to four bytes
- Improved and updated instructions for compatibility with Word variables
- New SX/B instructions
  -- Pin commands (SX48/52 only): **PULLUP**, **SCHMITT**, **CMOS**, **TLL**
  -- **COMPARE**
  -- **COUNT**
  -- **ON** *expression* **GOTO** | **GOSUB**
  -- **TIMER1** and **TIMER2** (SX48/52 only)
  -- **WDATA** directive

Version 1.42.01 -- Release date 17 OCT 2005

- Improved: **DATA** does not have to appear in listing before it is used
- Improved: **__PARAMx** storage for SX48/52 during interrupts
  -- SX48/52 now has 216 array elements
- Fixed: **LOOKUP** jump
- Fixed: **__REMAINDER** for SX18
- Fixed: **__PARAMCNT** for SX48/52
- Fixed: **RETURNINT** *var* for SX48/52

Version 1.42 -- Release date 29 AUG 2005

- Fixed: Error with *foo(bar) = SUBROUTINE_NAME*

Version 1.41 -- Release date 08 AUG 2005

- Fixed: Error with **>>** and **<<** operators.
- Fixed: Order of Label/String address values to offset, base
  -- now *Little Endian*
- Improved: Removed redundant instructions from **READ**.

Version 1.40 -- Release date 03 AUG 2005

- Improved: Baud rate performance for **SERIN**/**SEROUT**
- Added: Error raised if **SERIN**/**SEROUT** baud rate too high for target **FREQ**
- Added: Support for stored and inline strings (see **READ**)
- Fixed: foo(1) = foo(1) + bar(1).7
- Fixed: foo(1) = foo(1) - bar(1)
- Fixed: bitVar = foo(1).7
- Fixed Error: UNKNOWN COMMAND "LET" when using unknown variable name

- Fixed Error: When unknown variable is used with **PAUSEUS**

Version 1.31 -- Release date 08 JUL 2005

- Fixed: **INTURRUPT** and **RETURN** bug that affected SX48 and SX52
- Removed: *MOV W, #0* when **RETURN** used without value
- Improved: **PWM** *Duration* timing accuracy
- Added: Checks **FREQ** directive against **DEVICE** setting

Version 1.30 -- Release date 01 JUL 2005

- Added: **ID** directive
- Added: Subroutines can behave like functions, returning a value directly to a variable
  -- Example: *foo = SubName*
- **PAUSE** and **PAUSEUS** now allow fractional constants
- Allow *bitVar = byteVar*
  -- *bitVar* = 0 when *byteVar* = 0; *bitVar* = 1 when *byteVar* <> 0
- Fixed: *bitVar = ~bitVar* error

Version 1.22 -- Release date 9 MAY 2005

- Documented: **DJNZ**, and **SWAP**
- **BANK** directive updated -- may be used without parameters
- Added: **EXIT** for **FOR...NEXT** and **DO...LOOP**
- Fixed: Run-time error with **LOAD**

Version 1.21 -- Release date 27 APR 2005

- Added: **DO...LOOP**

Version 1.20 -- Release date 22 APR 2005

- Improved: SX/B supports all SX micros (SX18, SX20, SX28, SX48, and SX52)
- Added: Subroutine (**SUB**) declaration simplifies programming
- Added: Code pages handled automatically
- Added: **IF...THEN...ELSE...ENDIF**
- Added: Commands for Philips I²C® communications (master only)
- Added: Commands for Dallas/Maxim 1-Wire® communications
- Added: **SWAP** and **DJNZ** commands
- Added: **BANK** support (for SASM) -- simplifies access to non-global variables

Version 1.2 embodies major structual improvements and simplification over version 1.1, specifically in the area of declared subroutines and the automatic handling of code pages. Most programs written for version 1.1 will compile without change, however, it is recommended that these programs be updated to version 1.2 specifications to take full advantage of the SX/B compiler.

Version 1.10 -- Release date 15 DEC 2004

- Allow **BRANCH** to use @ with Label
- Allow **RETURNINT** *ByteVar*
- Added **FOR...NEXT** overflow checking
- Added **NOPRESERVE** option to **INTERRUPT**
- Allows bits to be passed as parameters (sets __PARAMx to 0 or 1)
- Added __RAM() system array (use with '@' RAM addresses)
- Allows computed constants (SASM only, not for SX/B commands)
- Improved timing accuracy of **PAUSE**, **PAUSEUS**, **PULSIN**, **PULSOUT**, **SERIN**, and **SEROUT**

Version 1.00 -- Release date 01 NOV 2004

- Initial release

Disclaimer of Liability

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

SX/B Definitions

I/O Pins

I/O pins are defined using the PIN definition -- either as a group, or as a single pin -- using the following syntax:

```
Symbol   PIN
Port{.Bit} {{OUTPUT}|{INPUT{SCHMITT|NOSCHMITT|CMOS|TTL} {PULLUP|NOPULLUP} {INTR_RISE|
INTR_FALL}}
```

```
DigCtrl     PIN RA OUTPUT                       ' 4 OUTPUT pins
Segments    PIN RB OUTPUT                       ' 8 OUTPUT pins
Sio         PIN RC 7 INPUT                      ' 1 INPUT pin
```

OUTPUT - Pin will be set to an output at startup

INPUT - Pin will be set to an input at startup

SCHMITT - Enables Schmitt trigger input levels

NOSCHMITT - Disables Schmitt trigger input levels

CMOS - Enables CMOS input levels (50% VDD)

TTL - Enables TTL input levels (1.4 V)

PULLUP - Enables built-in pullup resistors

NOPULLUP - Disables built-in pullup resistors

INTR_RISE - Enables interrupt on riding edge (Port RB only)

INTR_FALL - Enables interrupt on falling edge (Port RB only)

As of SX/B version 1.5, three 16-bit pseudo-ports have been added:

- RBC - ports RB and RC
- RCD - ports RC and RD (SX48/52)
- RDE - ports RD and RE (SX48/52)

On power-up or an external reset (via MCLR), an SX/B program clears all I/O pins (port bits cleared to 0) and initializes them to inputs (TRIS bits set to 1).

Constants

Constant values can be declared in four ways: decimal (default), hex, binary, and ASCII using the following syntax:

```
Symbol   CON   Value
```

Hex numbers are preceded with a dollar sign ($), binary numbers are preceded with a percent sign (%), and ASCII characters and strings are enclosed in double quotes ("). If no special punctuation is used then the SX/B compiler will assume the value is decimal. An underscore character may be used in numbers for clarity.

```
MaxCount    CON     4_000           ' decimal
LedMask     CON     $F0             ' hex
SegMap      CON     %0110_0101      ' binary
Letter      CON     "A"             ' ASCII character
Baud        CON     "T9600"         ' ASCII string
```

Constant names can be any combination of letters, numbers, and underscores (_), but the first character must not be a number. Also, constant names cannot use reserved words, such as SX/B instruction names (**SERIN**, **GOTO**, etc.) and SX aliases (RA, OPTION, etc.). The maximum number of characters allowed for a constant name is 32.

As of version 1.10 you may define computed constants for use by SASM in assembly routines. For example:

```
B2400       CON     16              ' 2400 Baud
B9600       CON     4               ' 9600 Baud
BitTm       CON     B9600           ' samples per bit
BitTm15     CON 3*BitTm/2           ' 1.5 bits
```

In the examples above, the first three constants may be used anywhere in the program. The final definition, for BitTm15, is a computed constant and my be used anywhere a variable is allowed. Note that computed constants may not be used where a constant value is required (e.g., the frequency parameter of FREQOUT).

String constants can be used to create "shortcuts" that ease programming, in effect, allowing the programmer to create new commands. For example:

```
GOSUB    RX_Byte, @serByte   ' get command
```

can be replaced with:

```
SERRX    @serByte        ' get command
```

by defining the following string constant:

```
SERRX CON  "GOSUB RX_Byte, "
```

Using string constants in this manner has generally been superceded by the **SUB** and **FUNC** definitions (for subroutine names and parameter requirements). See below for an improved method of declaring subroutines and functions.

Variables

SX/B supports words, bytes, arrays of bytes, and bit variables using the following syntax:

```
Symbol   VAR   Word{.bitIndex}
Symbol   VAR   Byte{(Size)}{.bitIndex}
Symbol   VAR   Bit
```

Variable space (bytes):

Device	General	Arrays	Max. Array
SX18/20	20	6x16 + 1x5 + 1x4	16 (each)
SX28	19	6x16 + 1x5 + 1x3	16 (each)
SX48/52	17	223	223

```
result      VAR   Word              ' word (16 bits)
count       VAR   Byte              ' one byte (eight bits)
display     VAR   Byte(4)           ' array of bytes
timer       VAR   Byte(NumTimers)   ' named constants are allowed
alarm       VAR   Bit               ' one bit
```

When defining byte arrays, only constant values or named constants may be used for the *Size* parameter, and array elements are zero-based, that is the elements are indexed from zero to Size-1. For example, myArray(10) contains the elements myArray(0) through myArray(9). In the SX18/20/28 arrays are limited to 16 elements; in the SX48/52 arrays may have up to 223 elements. When defining arrays for the SX18/20/28 it is best to define them largest to smallest to maximize RAM use efficiency.

Note that word variables may not be used in the index of an array. You can, however use the LSB of a word variable like this:

```
idx         VAR   Word

Start:
  FOR idx = 1 TO 10
    Leds(idx_LSB) = idx
  NEXT
  END
```

Variable names may be aliased (renamed) for programming convenience. This also allows a group of bit variables to be included in the same byte for single-line evaluation or modification

```
clock             VAR   Byte(3)     ' clock array
secs        VAR   clock(0)    ' seconds
mins        VAR   clock(1)    ' minutes
hrs         VAR   clock(2)    ' hours
flags             VAR   Word        ' all flags
hiTemp            VAR   flags.15    ' high temp flag
loTemp            VAR   flags.14    ' low temp flag
```

Internal SX aliases may also be renamed for improved program readability. For example:

```
Segs        PIN   RB          ' display segments (anodes)
TRIS_Segs   VAR   TRIS_B      ' segments TRIS reg
DigCtrl     PIN   RA          ' digit control (cathode)
TRIS_Dig    VAR   TRIS_A      ' digits TRIS reg
```

Variable names can be any combination of letters, numbers, and underscores (_), but the first character must not be a number. Also, variable names cannot use reserved words, such as SX/B instruction names (SERIN, GOTO, etc.) and SX aliases (RA, OPTION, etc.). The maximum number of characters allowed for a variable name is 32.

On power-up or an external reset (via MCLR), an SX/B program initializes all variables to zero unless the NOSTARTUP option is used with the PROGRAM directive.

Variables are usually passed by value using this form:

```
var1 = var2 ' copy var2 to var1
```

... actually copies the value of *var2* and places it into *var1*. The address (RAM location) of a variable may be passed by prefacing the variable name with '@':

```
var1 = @var2 ' put address of var2 into var1
```

This feature is particularly useful for passing address parameters to subroutines, especially when using **PUT**, **GET**, or the __RAM() system array.

Note that when assigning a bit variable the value of a byte or word, as in...

```
bitVar = variable
```

... the bit variable will get cleared to zero if the variable is zero, otherwise the bit variable will be set to one. If you would rather copy a specific bit from a byte or word variable, simply use the bit index you wish to copy:

```
bitVar = variable.bitIndex
```

Word variables have an additional assignment option with two byte values (variables or constants):

```
wordVar = lsbValue, msbValue
```

Program Labels

An SX/B program uses labels to define entry points into code sections or data tables. When used as a code entry point or data table name, label names must start in column one (not indented), end with a colon(:), and be on its own line. When used elsewhere in the program, labels are named without the colon, as in the example below:

```
Start:                                  ' colon required
  idx = 0
  PAUSE 500


Main:                                   ' colon required
  READINC Msg + idx, char               ' no colon required
  INC idx
  IF char = 0 THEN Start                ' no colon required
  SEROUT Sio, Baud, char
  GOTO Main                             ' no colon required

' ----------------------------------------------------------------------------


Msg:                                    ' colon required
  DATA "SX/B makes the SX fun!", 13, 0
```

Label names can be any combination of letters, numbers, and underscores (_), but the first character must not be a number. Also, label names cannot use reserved words, such as SX/B instruction names (SERIN, GOTO, etc.) and SX aliases (RA, OPTION, etc.). The maximum number of characters allowed for a label name is 32.

Comments

Comments can be used to add additional information to a program. The apostrophe character (') begins a comment section; anything to the right of the comment character will be ignored by the compiler. This allows comments to be added to a line of code. Using the comment character is also a convenient way to disable a line of code without removing it from the program.

```
' Display a running counter on RB

Main:
  INC RB                        ' update LEDs
'  INC RB                       ' this line disabled
  PAUSE 100                     ' insert delay
  GOTO Main                     ' run forever
```

Note: For backward-compatibility with older versions of the BASIC programming language, **REM** may be used to define a line comment, though this style is outdated and generally discouraged.

```
REM This is an old-fashioned style comment and typically not used.
```

Inline Assembly Instructions

SX assembly instructions can be inserted into an SX/B program using the "\" (back-slash) character to preface the assembly code statement. For large blocks of assembly code ASM...ENDASM is recommended.

```
LedsLo            PIN   RB OUTPUT
LedsHi            PIN   RC OUTPUT
cntr        VAR   Word              ' 16-bit counter

Start:

Main:
  \ MOV LedsLo, cntr_LSB            ' copy low byte to LEDs
  \ MOV LedsHi, cntr_MSB            ' copy high byte to LEDs
  PAUSE 100
  \ INC cntr_LSB                    ' update counter
  \ SZ                              ' skip if cntr_LSB is zero
  \ JMP Main                        ' jump to Main
  \ INC cntr_LSB                    ' increment high byte
  \ JMP Main                        ' jump to Main
```

Note: Array elements hold the address of a variable so they should not be used in inline assembly instructions.

Subroutine Declaration

The programming and use of subroutines is simplified by declaring the subroutine name and the parameter(s) (if any) required. Additionally, the declaration of subroutines allows them to return a byte value (see FUNC, below, for return word values). Declaring subroutines offers significant advantages to the programmer:

- The compiler can check code for the correct number of parameters
- **GOSUB** is no longer required to call the subroutine
- The subroutine can return a direct (byte) value
  -- This is simpler than passing the variable's address (@someVar)
- Code page management is automatically handled

SX/B subroutines are defined using the following syntax:

```
Label   SUB   {Min{, Max}}
```

Where *Min* is the minimum number of required parameters (if any) and *Max* is the maximum number of parameters passed to the subroutine. If *Max* is not specified then *Min* is the fixed number of parameters allowed.

The following short segment shows how predefined subroutines simplify SX/B program development:

```
TX_BYTE     SUB   1, 2              ' sub with 1 or 2 parameters

Start:
  TXBYTE "*"                       ' send one asterisk
  TXBYTE "-", 20                   ' send 20 dashes
  TXBYTE                           ' raises syntax error
  END

SUB TX_BYTE
  temp1 = __PARAM1                 ' save character
  IF __PARAMCNT = 2 THEN
    temp2 = __PARAM2               ' save repeats
  ELSE
    temp2 = 1                      ' set to 1 if not specified
  ENDIF
  DO WHILE temp2 > 0
    SEROUT SOut, Baud, temp1       ' send the byte
    DEC temp2                      ' dec repeats
  LOOP
  ENDSUB
```

Subroutines can also behave like functions in other languages, returning a byte value directly to a variable after the subroutine call. For example:

```
char = RX_BYTE
```

The value/variable to be returned to the calling code is placed after **RETURN** at the end of the subroutine code:

```
SUB RX_BYTE
  SERIN Sin, Baud, temp1            ' wait for serial input
  RETURN temp1                      ' return to caller
ENDSUB
```

When defining subroutines that require no parameters (as RX_BYTE, above), it is best to define the subroutine with a zero parameter count, as this will prevent the compiler from generating an assembly instruction (CLR __PARAMCNT) that is not needed by the program:

```
RX_BYTE SUB 0 ' receive serial byte
```

Subroutines do not have to be declared, but doing so allows the subroutine code to be placed anywhere in the listing (without declaration the code must be in the first half of a code page), **GOSUB** is no longer needed to call the subroutine, and the compiler is able to check the for the proper number of parameters. The only requirement is that the **SUB**declaration(s) be placed in the first half of a code page. The advantages of the **SUB** declaration far outweigh the minor effort required to add the declaration.

When declaring a subroutine for string handling, it must be set to accept at least two parameters (base and offset address values). See READ for details on handling strings with SX/B.

Function Declaration

As of SX/B 1.5, a subroutine can return one to four bytes when defined as a function. As with **SUB**, **FUNC** routines are declared in the first half of a code page, but the actual code may reside anywhere.

SX/B functions are defined using the following syntax:

```
Label   FUNC   ReturnCount{, Min{, Max}}
```

Where *ReturnCount* is the number of bytes (1 - 4) returned by the function, *Min* is the minimum number of required parameters (if any), and *Max* is the maximum number of parameters passed to the function.

The following short segment shows how to define and use a function that returns a 16-bit value:

```
FREQ_IN           FUNC  2           ' function returns two bytes

Start:
  freq1 = FREQ_IN ' get frequency
  END

FUNC FREQ_IN
  COUNT Fpin, 1000, tmpW1           ' count cycles for one second
  RETURN tmpW1                      ' return two bytes
ENDFUNC
```

Note that a function can return more bytes than the target variable. If, for example, a function is designed to return a word and the target output variable for that function is a byte, only the LSB of the return value will be assigned.

The programmer may assign additional return bytes manually, immediately following the function call. In the example below the function is designed to return a 32-bit result. The low word of the result is automatically assigned by the compiler, the high word of the result is manually assigned on the line that follows.

```
' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

result            VAR   Word        ' 32-bit result
resultHi    VAR   Word
tmpW1             VAR   Word        ' subroutine work vars
tmpW2             VAR   Word
tmpW3             VAR   Word
WATCH result, 32, UHEX              ' display 32-bit result

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

MULT32            FUNC  4, 2, 4

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  result = MULT32 $FFFF, $0010            ' get low word
  resultHi = __PARAM3, __PARAM4           ' get high word
  BREAK                             ' display result in Debug
  END

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' Use: MULT32 value1, value2
' -- multiplies two values
' -- when mixing a word and byte, the word must be declared first

FUNC MULT32
  IF __PARAMCNT = 2 THEN            ' byte * byte
    tmpW1 = __PARAM1
    tmpW2 = __PARAM2
  ENDIF
  IF __PARAMCNT = 3 THEN            ' word * byte
    tmpW1 = __WPARAM12
    tmpW2 = __PARAM3
  ENDIF
  IF __PARAMCNT = 4 THEN            ' word * word
    tmpW1 = __WPARAM12
    tmpW2 = __WPARAM34
  ENDIF
  tmpW3 = tmpW1 ** tmpW2            ' calculate high word
  tmpW2 = tmpW1 * tmpW2             ' calculate low word
  RETURN tmpW2, tmpW3               ' return 32 bits, LSW first
ENDFUNC
```

The discussion above applies to simple variables only. When using an array element as the target, all bytes are automatically assigned. For example:

```
bigVal            VAR   Byte(4)

...

  bigVal = MULT32 $1234, $1234
```

In this case, bigVal(0) .. bigVal(3) are assigned to the return variables __PARAM1 .. __PARAM4 from the function.

SX/B Directives

DEVICE (required)

DEVICE *[SX18 | SX20 | SX28 | SX48 | SX52] {, ...}*

The **DEVICE** directive specifies the device type (e.g, SX18, SX28), oscillator type, and other SX fuse settings.

```
DEVICE     SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
```

```
DEVICE     SX48, OSC4MHZ
```

Note that for the SX18, SX20, or SX28, the "TURBO", "STACKX", and "OPTIONX" options must be used. For the SX48 or SX52 they are not used.

Consult the *SX-Key Development System Manual* for a complete list of SX **DEVICE** options.

IRC_CAL

IRC_CAL*[IRC_SLOW | IRC_4MHZ | IRC_FAST]*

The **IRC_CAL** directive specifies the calibration value for the internal RC oscillator.

When the options IRC_SLOW or IRC_FAST are specified, the IRCTRIM bits in the FUSEX device configuration register are programmed to the minimum or maximum frequency value. When the option IRC_4MHZ is specified, the SX-Key software performs a calibration procedure whenever a program is downloaded to the SX chip and adjusts the IRCTRIM bits so that the internally generated clock frequency comes close as possible to 4 MHz.

```
IRC_CAL     IRC_4MHZ        ' calibrate internal RC clock
```

When not specified, the **IRC_CAL** setting is defaulted to IRC_SLOW.

FREQ

FREQ *Hertz{, EffectiveHz}*

The **FREQ** (frequency) directive is used to set the frequency (in Hertz) of the SX-Key's internal programmable oscillator to be used during debugging. **FREQ** is also used by the SX/B compiler for calculating delays in timing-sensitive instructions (**PAUSE**, **SERIN**, **SEROUT**, etc.), so connecting a clock source that differs from the compiled **FREQ** setting will affect stand-alone operation.

```
FREQ        4_000_000
```

Note that frequency can be any number from 31_250 to 110_000_000, but debugging via the SX-Key only works with frequencies between 400_000 and 11_0000_000. The underscore characters are used to help make the number more readable, as in 50_000_000, which is 50 MHz, but this convention is optional.

The optional parameter *EffectiveHz* is used to calculate the timing for SX/B instructions. If the program uses an interrupt to perform background functions this value can be used to compensate for the time spent in the interrupt routine. For example, if the interrupt is 100 cycles and is called every 1000 cycles, you would adjust the frequency by 10% by using:

```
FREQ        4_000_000, 3_600_000
```

ID

ID *ID_String*

The **ID** (identification) directive is used to write up to eight bytes of text into the ID word of the SX chip. This is used to record a version number or other unique identification for the code. This ID word can be read out of the SX chip at any time, regardless of the code protect setting. The line below will write GPXv2.1 into the ID word:

```
ID          "GPXv2.1"
```

WATCH

WATCH *Variable{.Bit}, Count, Format*

The **WATCH** directive allows the definition of format for viewing and modifying variables at runtime during debug mode. The variable's name, number of bits or bytes to view, and display format may be specified.

```
WATCH hertz, 16, UDEC
WATCH timer, 8, UDEC
WATCH flags.0, 1, UBIN
```

The table below lists the available format settings for the **WATCH** directive.

Format	Operation
UDEC	Displays value in unsigned decimal format
SDEC	Displays value in signed decimal format
UHEX	Displays value in unsigned hexadecimal format
SHEX	Displays value in signed hexadecimal format
UBIN	Displays value in unsigned binary format
SBIN	Displays value in signed binary format
FSTR	Displays value in fixed-length string format
ZSTR	Displays value in zero-terminated string format

Consult the *SX-Key Development System Manual* for additional information about the **WATCH** directive.

Note: As of SX/B version 1.5, **WATCH** may be used in a simplified format:

```
WATCH anyVariable
```

The compiler will insert the correct number of bits and specify UDEC format.

LOAD

LOAD *"FileName.SXB"*

The **LOAD** directive is used to insert an SX/B source code file at the current location.

```
LOAD "LCD.SXB"
```

INCLUDE

INCLUDE *"FileName"*

The **INCLUDE** directive is used to insert an SX assembly code file at the current location.

```
INCLUDE "I2C.SRC"
```

PROGRAM

PROGRAM *Label*{NOSTARTUP}

The **PROGRAM** directive sets the execution start point (at a label) for the SX/B program. Note that the **PROGRAM** directive must appear after the (optional) **INTERRUPT** hander, and in the first code page ($000 - $199).

When the **NOSTARTUP** option is used the SX/B compiler will not insert the normal start-up code that pre-initializes all RAM addresses to zero; in this case the programmer is responsible for appropriate initialization, except the FSR which is cleared (see **BANK**, below)

Note that when the **NOSTARTUP** option is used the TRIS registers are ***not*** initialized, so you ***must*** set up the ports as **INPUT**s or **OUTPUT**s.

```
PROGRAM Start
Start:
 OUTPUT RB

Main:
 INC RB
 PAUSE 1000
 GOTO Main
```

BANK

BANK {*DefaultBank***} {NOCODE}**

The **BANK** directive loads the BANK (FSR) variable with *DefaultBank*. The value will also be reloaded after any instruction that modifies BANK (FSR). *DefaultBank* may be a constant or a byte variable that is in the global RAM area (location <$10). If *DefaultBank* is the first variable declared, it will be in the global area.

If the NOCODE option is specified the BANK (FSR) is not changed immediately. This is useful if you know that the BANK (FSR) has ready been set, so there is no need to set it again. Since **BANK** is a compiler directive it's affect is strictly top-down, that is, its affect does not follow program flow if the program branches to a different section of code.

If **BANK** is used with no parameters, it will set the BANK (FSR) to the *DefaultBank*. This is useful in assembly routines, otherwise you would have to store the current FSR before using it.

BREAK

BREAK

The **BREAK** directive inserts a breakpoint into the assembly code which can useful during program debugging.

```
Start:
 OUTPUT RB BREAK

Main:
 INC RB
 GOTO Main
```

Note that only one **BREAK** directive may be used in an SX/B program. To create the effect of multiple breakpoints, the programmer may insert the **BREAK** directive into a subroutine that can be called from any point in the program.

ADDRESS (obsolete)

ADDRESS *PageAddr*

The **ADDRESS** directive sets the starting location for the instructions that follow.

```
Page_1:
 ADDRESS $200
```

Note that as of version 1.2, it is no longer necessary for the programmer to manually set code page addresses.

SX/B Operators

Unary Operators

- (Negate)

Variable = **-***Value*

Places the negative (two's complement) value of *Value* (which may be a constant or variable) into *Variable*.

```
Start:
 OUTPUT RB

Main:
 RB = $0F              ' RB = %00001111
 RB = -RB              ' RB = %11110001
 END
```

~, NOT (Bitwise Not)

Variable = **~***Value*
Variable = **NOT** *Value*

Places the bitwise inversion of *Value* (which may be a constant or variable) into *Variable*. Each result bit is subject to the following logic:

```
NOT 0 = 1
NOT 1 = 0
```

```
Start:
 OUTPUT RB

Main:
 xx = $F0              ' xx = %11110000
 RB = NOT xx             ' RB = %00001111
 RB = NOT RB             ' RB = %11110000
 RB = ~RB              ' RB = %00001111
 RB.0 = ~RB.0            ' RB = %00001110
 RB.1 = NOT xx.7           ' RB = %00001100
 END
```

Binary Operators

+ (Addition)

Variable = *Value1* **+** *Value2*

Adds two values (variables or constants) and places result in *Variable*. Note that when *Variable* is a byte the result will be truncated to eight bits.

```
Start:
  OUTPUT RB

Main:
  RB = $0F                      ' RB = %00001111
  RB = RB + $F0                 ' RB = %11111111, C = 0
  RB = RB + 2                   ' RB = %00000001, C = 1
  END
```

- (Subtraction)

Variable = *Value1* **-** *Value2*

Subtracts *Value2* from *Value1* and places the result in *Variable*. If the result is less than zero, two's-compliment format is used to store the result.

```
Start:
  OUTPUT RB

Main:
  RB = $0F                      ' RB = %00001111 (15)
  RB = RB - 7                   ' RB = %00001000 (8), C = 1
  RB = RB - 10                  ' RB = %11111110 (-2), C = 0
  END
```

* (Multiplication)

Variable = *Value1* ***** *Value2*

Multiplies *Value1* by *Value2* and places the result in *Variable*.

```
Start:
  OUTPUT RB

Main:
  RB = %00001111                ' RB = %00001111 (15)
  RB = RB * 16                  ' RB = %11110000 (240)
  RB = RB * 2                   ' RB = %11100000 (224)
  END
```

/ (Division)

Variable = *Value1* **/** *Value2*

Divides *Value1* by *Value2* and places the (integer) result in *Variable*.

```
Start:
  OUTPUT RB

Main:
  RB = $F0                    ' RB = %11110000 (240)
  RB = RB / 16                ' RB = %00001111 (15)
  RB = RB / 4                 ' RB = %00000011 (3)
  END
```

// (Modulus)

Variable = *Value1* **//** *Value2*

Divides *Value1* by *Value2* and places the remainder in *Variable*.

```
Start:
  OUTPUT RB

Main:
  RB = 25               ' RB = %00011001 (25)
  RB = RB // 10         ' RB = %00000101 (5)
  END
```

Note: The remainder of a division (modulus) is available immediately after the division or modulus operation. This code:

```
  dig10 = value / 10
  dig01 = __REMAINDER
```

... uses half the instruction space of:

```
  dig10 = value / 10
  dig01 = value // 10
```

The only requirement for using __REMAINDER (or __WREMAINDER for word values) is that the assignment to another variable must be the first instruction following the division or modulus operation.

*/ (Multiply Middle)

WordVar = *Value1* ***/** *Value2*

Multiplies *Value1* by *Value2*, returning the middle 16 bits of the 32-bit result to *WordVar*. This has the effect of multiplying a value by a whole number and a fraction. The whole number is the upper byte of the multiplier (0 to 255 whole units) and the fraction is the lower byte of the multiplier (0 to 255 units of 1/256 each).

The */ (star-slash) operator gives you an excellent workaround for SX/B's integer-only math. Suppose you want to multiply a value by 1.5. The whole number, and therefore the upper byte of the multiplier, would be 1, and the lower byte (fractional part) would be 128, since 128/256 = 0.5. It may be clearer to express the */ multiplier in hex -- as $0180 -- since hex notation keeps the contents of the upper and lower bytes separate.

```
WATCH tmpW1

Start:
  tmpW1 = 100
  tmpW1 = tmpW1 */ $0180            ' Multiply by 1.5 [1 + (128/256)]
  BREAK                             ' view in Debug Watch window
  END
```

** (Multiply High)

WordVar = *Value1* ****** *Value2*

Multiplies *Value1* by *Value2*, returning the high 16 bits of the result to *WordVar*. When you multiply two 16-bit values, the result can be as large as 32 bits. Since the largest variable supported by SX/B is 16 bits, the highest 16 bits of a 32-bit multiplication result are normally lost. The ** (star-star) operator gives you these upper 16 bits.

For example, suppose you multiply 65000 ($FDE8) by itself. The result is 4,225,000,000 or $FBD46240. The * (multiplication) operator would return the lower 16 bits, $6240; the ** operator returns $FBD4.

```
WATCH tmpW1, 16, UHEX
WATCH tmpW2, 16, UHEX

Start:
  tmpW1 = $FDE8
  tmpW2 = tmpW1 ** tmpW1            ' Multiply value1 by itself
  BREAK                             ' view in Debug Watch window
  END
```

MAX

Variable = *Value* **MAX** *Limit*

Places *Value* into *Variable* limiting the maximum value of *Variable* to *Limit*.

```
Start:
  OUTPUT RB

Main:
  FOR idx = 0 TO 15 ' idx = 0 .. 15
    RB = idx MAX 7  ' RB = 0 .. 7
    PAUSE 100
  NEXT idx
  GOTO Main
```

MIN

Variable = *Value* **MIN** *Limit*

Places *Value* into *Variable* forcing the minimum value of *Variable* to *Limit*.

```
Start:
  OUTPUT RB

Main:
  FOR idx = 0 TO 15           ' idx = 0 .. 15
    RB = idx MIN 7            ' RB = 7 .. 15
    PAUSE 100
  NEXT idx
  GOTO Main
```

&, AND (Bitwise AND)

Variable = *Value1* **&** *Value2*
Variable = *Value1* **AND** *Value2*

Performs a bitwise AND operation on Value1 and Value2, then places the result in *Variable*. Each result bit is subject to the following logic:

```
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

```
Start:
  OUTPUT RB

Main:
  RB = $0F & %11111111          ' RB = %00001111
  RB = RB AND %11111100         ' RB = %00001100
  END
```

|, OR (Bitwise OR)

Variable = *Value1* **|** *Value2*
Variable = *Value1* **OR** *Value2*

Performs a bitwise OR operation on Value1 and Value2, then places the result in *Variable*. Each result bit is subject to the following logic:

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

```
Start:
  OUTPUT RB

Main:
  RB = %00001111              ' RB = %00001111
  RB = RB | %11000000         ' RB = %11001111
  RB = RB OR %00011111        ' RB = %11011111
  END
```

^, XOR (Bitwise Exclusive OR

Variable = *Value1* **^** *Value2*
Variable = *Value1* **XOR** *Value2*

Performs a bitwise Exclusive OR operation on Value1 and Value2, then places the result in *Variable*. Each result bit is subject to the following logic:

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
```

```
Start:
  OUTPUT RB

Main:
  RB = %00001111              ' RB = %00001111
  RB = RB ^ %11000000         ' RB = %11001111
  RB = RB XOR %00011111       ' RB = %11010000
  END
```

<<, SHL (Shift Left)

Variable = *Value1* **<<** *Value2*
Variable = *Value1* **SHL** *Value2*

Left shifts the bits of *Value1* by the number specified in *Value2*. Bits shifted off the left end (MSB) of a number are lost; bits shifted into the right end of the number are zeros. Shifting the bits of a value left n number of times has the same effect as multiplying that number by 2 to the nth power. For instance 15 << 3 (shift the bits of the decimal number 15 left three places) is equivalent to 15 * 2^3 (15 * 8).

```
Start:
  OUTPUT RB

Main:
  RB = %00001101 << 2         ' RB = %00110100
  RB = RB SHL 4               ' RB = %01000000
  END
```

>>, SHR (Shift Right)

Variable = *Value1* **>>** *Value2*
Variable = *Value1* **SHR** *Value2*

Right shifts the bits of *Value1* by the number specified in *Value2*. Bits shifted off the right end (LSB) of a number are lost; bits shifted into the left end of the number are zeros. Shifting the bits of a value right n number of times has the same effect as dividing that number by 2 to the nth power. For instance $F0 >> 3 (shift the bits of the decimal number 240 right three places) is equivalent to 240 / 2^3 (240 / 8).

```
Start:
  OUTPUT RB

Main:
  RB = %10110000 >> 2          ' RB = %00101100
  RB = RB SHR 4          ' RB = %00000010
  END
```

SX/B Aliases

SX Registers

			SX18	SX20	SX28	SX48	SX52
• IND	Indirect Register	$00	✓	✓	✓	✓	✓
• RTCC	Real Time Clock/Counter	$01	✓	✓	✓	✓	✓
• PC	Program Counter	$02	✓	✓	✓	✓	✓
• STATUS	Status Register	$03	✓	✓	✓	✓	✓
• PA2	Page Select bit 2	STATUS.7	✓	✓	✓	✓	✓
• PA1	Page Select bit 1	STATUS.6	✓	✓	✓	✓	✓
• PA0	Page Select bit 0	STATUS.5	✓	✓	✓	✓	✓
• _TO	Time Out	STATUS.4	✓	✓	✓	✓	✓
• PD	Power Down	STATUS.3	✓	✓	✓	✓	✓
• Z_	Zero	STATUS.2	✓	✓	✓	✓	✓
• DC	Digit Carry	STATUS.1	✓	✓	✓	✓	✓
• C	Carry	STATUS.0	✓	✓	✓	✓	✓
• FSR	File Select Register	$04	✓	✓	✓	✓	✓
• RA	Port A	$05	✓	✓	✓	✓	✓
• RB	Port B	$06	✓	✓	✓	✓	✓
• RC	Port C	$07	✗	✗	✓	✓	✓
• RD	Port D	$08	✗	✗	✗	✓	✓
• RE	Port E	$09	✗	✗	✗	✓	✓
• RBC	Ports B & C as 16-bit entity	$06 (LSB) + $07 (MSB)	✗	✗	✓	✓	✓
• RCD	Ports C & D as 16-bit entity	$07 (LSB) + $08 (MSB)	✗	✗	✗	✓	✓
• RDE	Ports D & E as 16-bit entity	$08 (LSB) + $09 (MSB)	✗	✗	✗	✓	✓
• PORTA	Port A	$05	✓	✓	✓	✓	✓
• PORTB	Port B	$06	✓	✓	✓	✓	✓
• PORTC	Port C	$07	✗	✗	✓	✓	✓
• PORTD	Port D	$08	✗	✗	✗	✓	✓
• PORTE	Port E	$09	✗	✗	✗	✓	✓
• TRIS_A	Data Direction register for RA		✓	✓	✓	✓	✓
• TRIS_B	Data Direction register for RB		✓	✓	✓	✓	✓
• TRIS_C	Data Direction register for RC		✗	✗	✓	✓	✓
• TRIS_D	Data Direction register for RD		✗	✗	✗	✓	✓
• TRIS_E	Data Direction register for RE		✗	✗	✗	✓	✓
• PLP_A [1]	Pull-Up resistor enable for RA		✓	✓	✓	✓	✓
• PLP_B [1]	Pull-Up resistor enable for RB		✓	✓	✓	✓	✓
• PLP_C [1]	Pull-Up resistor enable for RC		✗	✗	✓	✓	✓
• PLP_D [1]	Pull-Up resistor enable for RD		✗	✗	✗	✓	✓
• PLP_E [1]	Pull-Up resistor enable for RE		✗	✗	✗	✓	✓
• LVL_A [1]	TTL/CMOS select for RA		✓	✓	✓	✓	✓
• LVL_B [1]	TTL/CMOS select for RB		✓	✓	✓	✓	✓
• LVL_C [1]	TTL/CMOS select for RC		✗	✗	✓	✓	✓
• LVL_D [1]	TTL/CMOS select for RD		✗	✗	✗	✓	✓
• LVL_E [1]	TTL/CMOS select for RE		✗	✗	✗	✓	✓
• ST_B [1]	Schmitt-Trigger enable for RB		✓	✓	✓	✓	✓
• ST_C [1]	Schmitt-Trigger enable for RC		✗	✗	✓	✓	✓
• ST_D [1]	Schmitt-Trigger enable for RD		✗	✗	✗	✓	✓
• ST_E [1]	Schmitt-Trigger enable for RE		✗	✗	✗	✓	✓
• WKEN_B [1]	Wake Up enable register		✓	✓	✓	✓	✓
• WKED_B [1]	Wake Up edge select register		✓	✓	✓	✓	✓
• WKPND_B [1, 2]	MIWU pending register		✓	✓	✓	✓	✓

• CMP_B [1, 2]	Comparator enable register	✓	✓	✓	✓	✓
• T1CPL	Low byte of Timer T1 capture register	✗	✗	✗	✓	✓
• T1CPH	High byte of Timer T1 capture register	✗	✗	✗	✓	✓
• T1R1CML	Low byte of Timer T1 compare register 1	✗	✗	✗	✓	✓
• T1R1CMH	High byte of Timer T1 compare register 1	✗	✗	✗	✓	✓
• T1R2CML	Low byte of Timer T1 compare register 2	✗	✗	✗	✓	✓
• T1R2CMH	High byte of Timer T1 compare register 2	✗	✗	✗	✓	✓
• T1CNTA	Timer T1 control register A	✗	✗	✗	✓	✓
• T1CNTB	Timer T1 control register B	✗	✗	✗	✓	✓
• T2CPL	Low byte of Timer T2 capture register	✗	✗	✗	✓	✓
• T2CPH	High byte of Timer T2 capture register	✗	✗	✗	✓	✓
• T2R1CML	Low byte of Timer T2 compare register 1	✗	✗	✗	✓	✓
• T2R1CMH	High byte of Timer T2 compare register 1	✗	✗	✗	✓	✓
• T2R2CML	Low byte of Timer T2 compare register 2	✗	✗	✗	✓	✓
• T2R2CMH	High byte of Timer T2 compare register 2	✗	✗	✗	✓	✓
• T2CNTA	Timer T2 control register A	✗	✗	✗	✓	✓
• T2CNTB	Timer T2 control register B	✗	✗	✗	✓	✓
• MODE	Mode register	✓	✓	✓	✓	✓
• M	Mode register	✓	✓	✓	✓	✓
• OPTION	Option register	✓	✓	✓	✓	✓

[1] When changed (i.e., *Register* = *Value*), the SX/B compiler automatically inserts the correct Mode (M) register instruction.
[2] When changed (i.e., *Register* = *Variable*), *Variable* is swapped with *Register*.

SX/B Variables

			SX18	SX20	SX28	SX48	SX52
• __RAM()	Provides access to any RAM address	$00	✓	✓	✓	✓	✓
• __PARAM1	First byte parameter	$08 / $0A *	✓	✓	✓	✓	✓
• __PARAM2	Second byte parameter	$09 / $0B *	✓	✓	✓	✓	✓
• __PARAM3	Third byte parameter	$0A / $0C *	✓	✓	✓	✓	✓
• __PARAM4	Fourth byte parameter	$0B / $0E *	✓	✓	✓	✓	✓
• __WPARAM12	First word parameter	$08,09 / $0A,$0B *	✓	✓	✓	✓	✓
• __WPARAM23	Second word parameter	$09,$0A / $0B,$0C *	✓	✓	✓	✓	✓
• __WPARAM34	Third word parameter	$0A,$0B / $0C,$0D *	✓	✓	✓	✓	✓
• __REMAINDER	Remainder after a division	$08 / $0A *	✓	✓	✓	✓	✓
• __WREMAINDER	Remainder after a division	$08,$09 / $0A,$0B *	✓	✓	✓	✓	✓
• __PARAMCNT	Number of parameters passed to subroutine	$0C / $0E *	✓	✓	✓	✓	✓
• __INTPARAMFSR	Save/Restore location of "M" and __PARAMx	$F5	✓	✓	✓	✓	✓
• __TRISA †	TRIS_A copy	$FA	✓	✓	✓	✓	✓
• __TRISB †	TRIS_B copy	$FB	✓	✓	✓	✓	✓
• __TRISC †	TRIS_C copy	$FC	✗	✗	✓	✓	✓
• __TRISD †	TRIS_D copy	$FD	✗	✗	✗	✓	✓
• __TRISE †	TRIS_E copy	$FE	✗	✗	✗	✓	✓

SX/B Constants

		SX18	SX20	SX28	SX48	SX52
• __FREQMHZ	Frequency (in MHz) as set in FREQ directive.	✓	✓	✓	✓	✓

* **Note**: Location depends on module used: SX18/20/28 or SX48/52

† **Note**: These registers must not be modified by the programmer, otherwise some SX/B instructions will be adversely affected.

SX/B Commands

- All SX chips
- SX48/52 only

ANALOGIN	**ANALOGIN** *InPin*, *OutPin*, *Result* {, *Prime* }
ASM	**ASM** *Instruction(s)* ... **ENDASM**
BRANCH	**BRANCH** *Offset*, *Label0* {, *Label1, Label2, ...}*
CLEART1	Obsolete - replaced with TIMER1 CLEAR
CLEART2	Obsolete - replaced with TIMER2 CLEAR
CMOS	**CMOS** *Pin* {, *Enable}*
COMPARE	**COMPARE** *Mode, Result*
COUNT †	**COUNT** *Pin*, *Duration*, *Variable*
DATA	**DATA** *Const0* {, *Const1, Const2, ...}*
DEC	**DEC** *Variable*
DJNZ	**DJNZ** *Variable, Label*
DO ... LOOP	**DO** {**WHILE** | **UNTIL***Condition* } ... **LOOP** {**NEVER** | **UNTIL** | *Condition* | **WHILE***Condition* }
END	**END**
EXIT	{**IF** *Condition* **THEN**} **EXIT**
FOR ... NEXT	**FOR** *ByteVar* = *StartVal* **TO** *EndVal*{**STEP** {-}*StepVal*} ... **NEXT**
FREQOUT †	**FREQOUT** *Pin*, *Duration*, *Freq*
GET	**GET** *Location, ByteVar* {, *ByteVar, ...}*
GOSUB ... RETURN	**GOSUB** *Label* ... **RETURN** *{ Value }*
GOTO	**GOTO** *Label*
HIGH	**HIGH** *Pin*
I2CRECV †	**I2CRECV** *Pin*, *ByteVar*, *AckBit*
I2CSEND †	**I2CSEND** *Pin*, *ByteVal* {, *AckVar*}
I2CSTART †	**I2CSTART** *Pin*
I2CSTOP †	**I2CSTOP** *Pin*
IF ... THEN	**IF** *Condition* [**THEN** | **GOTO**] [*Label* | **EXIT**]
IF ... THEN ... ELSE	**IF** *Condition* **THEN** *Statement(s)* { **ELSE** *Statement(s)* } **ENDIF**
INC	**INC** *Variable*
INPUT	**INPUT** *Pin*
INTERRUPT	**INTERRUPT** *Instruction(s)* ... **RETURNINT** *{Cycles}*
LET	{**LET**} *Expression*
LOOKDOWN	**LOOKDOWN** *Target, Value0, {Value1, Value2, ...} Variable*
LOOKUP	**LOOKUP** *Index, Value0, {Value1, Value2, ...} Variable*
LOW	**LOW** *Pin*
ON	**ON** *Expression* [**GOTO** | **GOSUB**] *Label0* {, *Label1, Label2}*
OUTPUT	**OUTPUT** *Pin*
OWRDBIT †	**OWRDBIT** *Pin, BitVar*
OWRDBYTE †	**OWRDBYTE** *Pin, ByteVar*
OWRESET †	**OWRESET** *Pin* {, *ByteVar}*
OWWRBIT †	**OWWRBIT** *Pin, BitVal*
OWWRBYTE †	**OWWRBYTE** *Pin, ByteVal*
PAUSE †	**PAUSE** *Value1 {[, | *] Value2}*
PAUSEUS †	**PAUSEUS** *Value1 {[, | *] Value2}*
PULLUP	**PULLUP** *Pin* {, *Enable}*

PULSIN †	**PULSIN** *Pin, State, ByteVar {, Resolution}*
PULSOUT †	**PULSOUT** *Pin, Duration {, Resolution}*
PUT	**PUT** *Location, Value {, Value, ...}*
PWM †	**PWM** *Pin, Duty, Duration*
RANDOM	**RANDOM** *Seed {, Duplicate}*
RCTIME †	**RCTIME** *Pin, StartState, ByteVar {, Resolution}*
READ / READINC	**READ** | **READINC** *Base {+ Offset}, ByteVar {, ByteVar, ...}*
RESETWDT	**RESETWDT**
REVERSE	**REVERSE** *Pin*
SCHMITT	**SCHMITT** *Pin {, Enable}*
SERIN †	**SERIN** *Pin, BaudMode, ByteVar {, Timeout, Label}*
SEROUT †	**SEROUT** *Pin, BaudMode, Value*
SHIFTIN	**SHIFTIN** *DPin, CPin, ShiftMode, ByteVar {|Count}*
SHIFTOUT	**SHIFTOUT** *DPin, CPin, ShiftMode, Value {|Count}*
SLEEP	**SLEEP**
SOUND †	**SOUND** *Pin, Note, Duration*
SWAP	**SWAP** *Variable*
TIMER	**TIMER[1 | 2]** *Command {Value {, Value}}*
TOGGLE	**TOGGLE** *BitVar*
TTL	**TTL** *Pin {, Enable}*
WDATA	**WDATA** *Const0 {, Const1, Const2, ...}*

†Note: While all SX/B commands will run at any FREQ setting, commands that are time-sensitive (particularly PAUSEUS) have been designed for FREQ settings between 4 MHz and 50 MHz. FREQ settings outside this range (e.g. low-power applications running at 32 kHz) are not recommended when using time-sensitive instructions.

ANALOGIN

Example

ANALOGIN *InPin, OutPin, Result {, Prime}*

Function

The SX/B **ANALOGIN** command converts an analog voltage to a digital value by using a method known as continuous calibration. This method requires one input pin and one output pin on the SX chip. The only required hardware is two resistors and one capacitor.

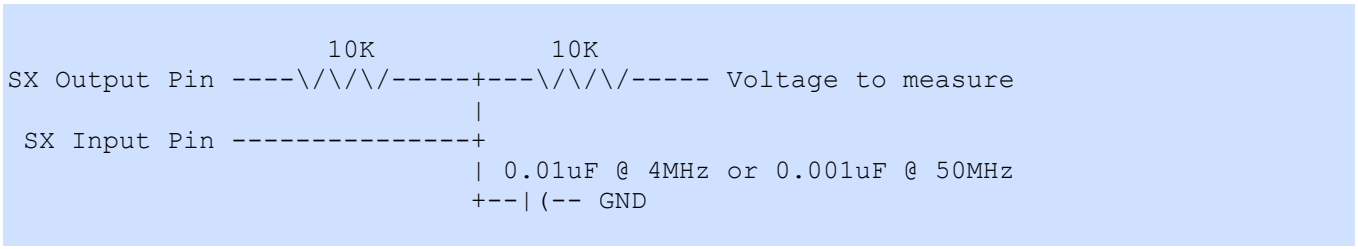
- ***InPin*** is the input pin.
- ***OutPin*** is the output pin.
- ***Result*** is a byte variable that will receive the value.
- ***Prime*** is the number of priming cycles to execute before taking the measurement. Prime is optional; if not specified the default is 1 cycle.

Explanation

The method works by taking advantage of the input pin's threshold voltage. This is the voltage level that makes the input pin read as either a "0" or a "1". Normally on the SX the input threshold is set to the "TTL" level, which is 1.4 volts. So voltages above 1.4 volts are read as a "1" and voltages below 1.4 volts are read as a "0". To allow the measured voltage to range from 0 volts to Vdd we need to set the pin to the "CMOS" threshold level, which is 1/2 Vdd (or 2.5 volts when operating the SX from a 5 volt supply).

```
' Example program using ANALOGIN command


DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000

InPin             PIN   RA.0 INPUT CMOS
OutPin            PIN   RA.1 OUTPUT


               10K            10K
'RA.1 Pin ----\/\/\/-----+---\/\/\/----- Voltage to measure
'                        |
'RA.0 Pin ---------------+
'                        | 0.01uF
'                        +--|(-- GND

a           VAR   Byte

PROGRAM Start NOSTARTUP

Start:
  ANALOGIN InPin, OutPin, a, 2
  WATCH a
  BREAK
  GOTO Start
END
```

For this example we will assume the SX is operating from 5 volts, and that you have set the input pin to "CMOS" threshold levels (the easiest way to do this is to use the PIN definition "InPin PIN INPUT CMOS"). Here is how the components are connected:



What the **ANALOGIN** command does is read the input pin, and make the output pin the opposite of what the input pin reads. If the input pin reads "0", it makes the output pin a "1". If the input pin reads "1", it makes the output pin a "0". It does this 255 times, and keeps a count of how many times the input pin was a "1". This count is what is returned as the result of the command. This value is proportional to the voltage level.

Basically the ANALOGIN command attempts to keep the input pin right at the threshold voltage. If the voltage input was not connected, and the capacitor wasn't there, the output would just toggle from high to low. And the count would end up being 128. Now if you added the cap, the output pin would still toggle, but not every time (since it takes the cap some time to charge and discharge), but over the long run it would still return a count of 128.

Now imagine if you have the complete circuit connected and the voltage input is 0 volts. The input pin will read as a "0" so it will make the output pin high (5 volts). So now we have 0 volts through a 10K resistor and 5 volts through a 10K resistor. That will make the junction (where the cap and input pin are connected) equal 2.5 volts. So the input pin will never get above 2.5 volts regardless of how long the output pin stays high, so it will always read as a "0" and our count will be zero.

Now imagine if the input voltage is 5 volts. The input will read as a "1", so it will make the output low (0 volts). Now we have the same situation reversed. The voltage at the input pin can never get below 2.5 volts regardless of how long the output pin stays low, so it will always read as a "1" and our count will be 255.

When the input voltage is between 0 volts and 5 volts, then things get interesting. If the input voltage is 1.25 volts (1/4 the maximum), then the input pin will see a pattern of 0's and 1's such that the number of 0's is 3 times the number of 1's. Over 255 samples it will return a count of 63 (since only 1 in 3 reads of the input pin were 1's). Depending on the clock speed of the SX and the value of the capacitor, the pattern may be something like "000100010001" or it may be something like "000000110000001100000011". But over the 255 samples you will still get a count of about 63 1's in the pattern.

It may take some experimentation to get the optimum values for the capacitor. In general the faster the SX clock, the lower the capacitor value, and the slower the SX clock, the higher the capacitor value.

Another factor that affects the stability of **ANALOGIN** is that the method assumes the input pin is already at the threshold voltage before it starts counting the 1's read at the input pin. To accomplish this the **ANALOGIN** command actually primes the capacitor by running 255 samples BEFORE starting to count the pulses. Then it runs another 255 samples while counting the 1's. There is an optional parameter that can be used with the **ANALOGIN** command if you want or need more priming cycles (255 samples per cycle). More priming cycles allow the use of a larger capacitor and that gives more stable readings, but takes more time to complete the **ANALOGIN** command. So if you can afford the extra time, and want a more stable reading, then increase the value of the capacitor and increase the number of priming cycles.

Okay so what if you want to read voltage ranges other than 0 volts to 5 volts. Well if you want get full scale values from a voltage lower than 5 volts, one easy way is to just set the input pin to it's default setting of "TTL" threshold levels. Since the "TTL" threshold level is 1.4 volts, and the **ANALOGIN** values range from 0 volts to 2x the threshold level, this will result in 0 for 0 volts and 255 for 2.8 volts. Wider voltage ranges can be read by using asymmetrical resistor values. If you make the resistor connected to the measured voltage a larger value than the resistor connected to the SX output pin, you can read voltages greater than 5 volts.

Note that the values returned by **ANALOGIN** will be dependent on the impedance of the voltage being measured. The resistors used should be several times larger than the impedance of the input voltage. For example if you were using a 10K pot to create voltage from 0 to 5 volts the resistance of the pot would effectively be added to the 10K resistor. This would make the resistor values unequal. Let's suppose the pot was centered. The output of the pot would equate to a 2.5K resistor connected directly to a 2.5 volt supply. This 2.5K resistance would be in series with the 10 k resistor connected from the pot to the cap. Since the pot would have effectively zero resistance when turned to each end point, you would still get the full range of values, but the values would not be linear through the range of the pot.

Related instructions: IF ... THEN and ON

ASM...ENDASM

ASM *Instruction(s)* . . . **ENDASM**

Function

ASM allows the insertion of a block of assembly language statements into the SX/B program. The assembly language block is terminated with **ENDASM**. Code in the **ASM..ENDASM** block is inserted into the program verbatim.

Explanation

Certain time-critical routines are best coded in straight assembly language, and while the **** symbol allows the programmer to insert a single line of assembly code, it is not convenient for large blocks.

```
' Use: inByte = SHIFTIO outByte
' -- sends (via ShOut) and receives (via ShIn) bytes LSBFIRST

SHIFTIO:
   \ CLR tmpB1                      ' clear input byte
   \ MOV idx, #8                    ' do 8 bits


ShIO_Loop:
  ASM    MOVB ShOut, __PARAM1 0     ' move LSB out to pin
    MOV __PARAM3, #50               ' 50 us pause @ 4 MHz
    DJNZ __PARAM3, $
    XOR RA, #%00000001              ' toggle clock
    MOV __PARAM3, #50
    JNZ __PARAM3, $
    CLC
    RR tmpB1                        ' prep for input bit
    MOVB tmpB1.7, ShIn              ' capture input bit (LSB)
    XOR RA, #%00000001              ' toggle clock
    CLC
    RR __PARAM1                     ' prep for next output bit
    DJNZ idx, ShIO_Loop             ' repeat for 8 bits
  ENDASM  RETURN tmpB1
```

Related example: INTERRUPT Examples

BRANCH

Example

BRANCH *Offset, Label0 {, Label1, Label2, ...}*

Function

Jump to the program ***Label*** specified by ***Offset*** which can be a Byte or Word variable. Note that the value of ***Offset*** should not be greater than the number of labels-1, otherwise the **BRANCH** instruction will be skipped.

- ***Offset*** is a Byte or Word variable that specifies the index of the address label, in the list, to branch to. If the ***Offset*** exceeds the number of labels, the program will continue at the line following **BRANCH**. When ***Offset*** is a Word, only the LSB is used.
- ***Labels*** specify the possible targets for the **BRANCH** instruction.

Explanation

The **BRANCH** instruction is useful when you want to do something like this:

```
Test_Value:
  IF value = 0 THEN Case_0    ' value = 0: go to label "Case_0"
  IF value = 1 THEN Case_1    ' value = 1: go to label "Case_1"
  IF value = 2 THEN Case_2    ' value = 2: go to label "Case_2"
  IF value = 3 THEN Case_3    ' value = 3: go to label "Case_3"
  IF value = 4 THEN Case_4    ' value = 4: go to label "Case_4"
```

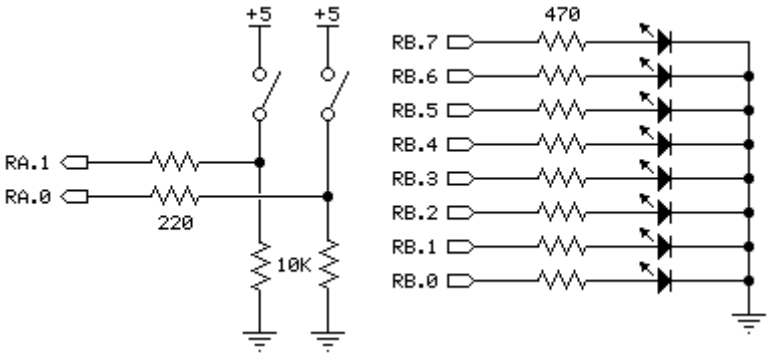
You can convert a long list of **IF-THEN** statements to **BRANCH** to like this:

```
Test Value:
  BRANCH value, Case_0, Case_1, Case_2, Case_3, Case_4

No_Branch:
```

Related instructions: IF ... THEN and ON

BRANCH Example

Syntax




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Controls eight channels of lights using RB. Sequence is selected by
' switches connected to RA.0 and RA.1. The delay between steps is fixed
' by a constant, but could easily be modified to be variable by using
' RCTIME.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "BRANCH"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

Select            PIN   RA INPUT          ' bits 0 and 1
Lights            PIN   RB OUTPUT

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

StepDelay CON 100                         ' 100 ms between steps

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

choice            VAR   Byte              ' selected sequence
maxSteps    VAR   Byte                    ' steps in sequence
idx         VAR   Byte                    ' step pointer

' =============================================================================
 PROGRAM Start
```

```
' =============================================================================

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  LOW Lights                              ' clear lights

Main:
  PAUSE StepDelay                         ' inter-step delay
  choice = Select & %0011                 ' get show
  BRANCH choice, Show_0, Show_1, Show_2, Show_3
  GOTO Main

Show_0:
  READ Pattern0, maxSteps                 ' get steps in sequence
  IF idx <= maxSteps THEN                 ' check idx range
    READINC Pattern0 + idx, Lights        ' get next step pattern
    INC idx                               ' point to next step
  ELSE
    idx = 1                               ' reset idx if needed
  ENDIF
  GOTO Main

Show_1:
  READ Pattern1, maxSteps
  IF idx <= maxSteps THEN
    READINC Pattern1 + idx, Lights
    INC idx
  ELSE
    idx = 1
  ENDIF
  GOTO Main

Show_2:
  READ Pattern2, maxSteps
  IF idx <= maxSteps THEN
    READINC Pattern2 + idx, Lights
    INC idx
  ELSE
    idx = 1
  ENDIF
  GOTO Main

Show_3:
  READ Pattern3, maxSteps
  IF idx <= maxSteps THEN
    READINC Pattern3 + idx, Lights
    INC idx
  ELSE
    idx = 1
  ENDIF
  GOTO Main

' =============================================================================
' User Data
' =============================================================================

Pattern0:
```

```
  DATA 8                                ' steps in sequence
  DATA %00000001                        ' sequence values
  DATA %00000010
  DATA %00000100
  DATA %00001000
  DATA %00010000
  DATA %00100000
  DATA %01000000
  DATA %10000000

Pattern1:
  DATA 10
  DATA %00000000
  DATA %10000000
  DATA %11000000
  DATA %01100000
  DATA %00110000
  DATA %00011000
  DATA %00001100
  DATA %00000110
  DATA %00000011
  DATA %00000001

Pattern2:
  DATA 5
  DATA %00000000
  DATA %00011000
  DATA %00100100
  DATA %01000010
  DATA %10000001

Pattern3:
  DATA 5
  DATA %11111111
  DATA %01111110
  DATA %00111100
  DATA %00011000
  DATA %00000000
```

CMOS (SX48/52 Only)

CMOS *Pin {, Enable}*

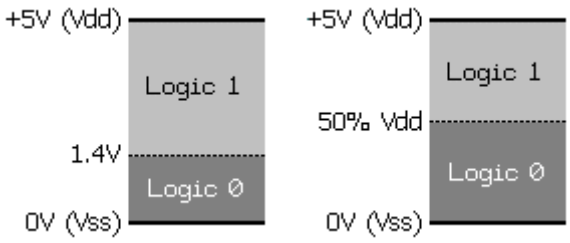
Function

Configures ***Pin*** for CMOS input threshold (50% of Vdd) on the SX48 or SX52. This command does not apply to the SX18, SX20, or SX28 (use the LVL_A, LVL_B, and LVL_C registers).

- ***Pin*** is any SX48/52 I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***Enable*** is a constant, 0 or 1, that enables (1) or disables (0) the CMOS input threshold. When not specified, *Enable* defaults to 1. If *Enable* is 0, the pin will be configured for TTL input threshold.

Explanation

Every I/O pin has selectable logic level control that determines the voltage threshold for a logic level 0 or 1. The default logic level for all I/O pins is TTL but can be modified by writing to the appropriate logic-level register (LVL_A, LVL_B, LVL_C, LVL_D and LVL_E). The logic level can be configured for all pins, regardless of pin direction, but really matters only when the associated pin is set to input mode. By configuring logic levels on input pins, the SX chip can be sensitive to both TTL and CMOS logic thresholds. The figure below demonstrates the difference between TTL and CMOS logic levels.



TTL Logic Level **CMOS Logic Level**

The logic threshold for TTL is 1.4 volts; a voltage below 1.4 is considered to be a logic 0, while a voltage above is considered to be a logic 1. The logic threshold for CMOS is 50% of Vdd, a voltage below ½ Vdd is considered to be a logic 0, while a voltage above ½ Vdd is considered to be a logic 1.

```
Start:
  CMOS RE.7, 1          ' set to CMOS level
  CMOS RE.6             ' set to CMOS level
  CMOS RE.5, 0          ' disable CMOS level, set to TTL level
```

Related instructions: TTL, PULLUP, and SCHMITT

COMPARE

Example

COMPARE *Mode, Result*

Function

Enable or disable the SX comparator, compare voltages on RB.1 and RB.2, and retrieve comparison result to store in ***Result***.

- ***Mode*** is a variable or constant (0 – 2) that enables or disables the comparator (RB.1 and RB.2) and determines if the optional comparator output pin (RB.0) is enabled or not. See the table below for an explanation of the ***Mode*** values.
- ***Result*** is a variable (usually a bit) in which the comparison result is stored.

Quick Facts

	SX18 / SX20 / SX28 / SX48 / SX52
Mode values	0: Disables comparator. 1: Enables comparator with RB.0 as *Result* output. 2: Enables comparator without RB.0 as *Result* output.
Result values	0: Voltage RB.1 > RB.2; RB.0 optionally outputs 0. 1: voltage RB.1 < RB.2; RB.0 optionally outputs 1.

Explanation

The **COMPARE** instruction enables or disables the built-in comparator hardware on the SX's pins RB.0, RB.1, and RB.2. I/O pins RB.1 and RB.2 are the comparator inputs and RB.0 is, optionally, the comparator result output pin.

By default, the comparator feature is disabled. Using the **COMPARE** instruction with a *Mode* argument of 1 or 2 enables the comparator feature (using input pins RB.1 and RB.2) and returns the result of the comparison in *Result*. If *Mode* is 1, the result of the comparison is also output on I/O pin RB.0. The following is an example of the **COMPARE** instruction:

```
COMPARE 1, result ' enable comparator, output result on RB.0
```

This example enables the comparator (setting RB.0 to output the result, with RB.1 and RB.2 as the comparator inputs) and writes the result of the comparison into *result*. Both *result* and the output pin RB.0 will be 0 if the input voltage on RB.1 was greater than that of RB.2. *Result* and the output pin RB.0 will be 1 if the input voltage on RB.1 was less than that of RB.2.

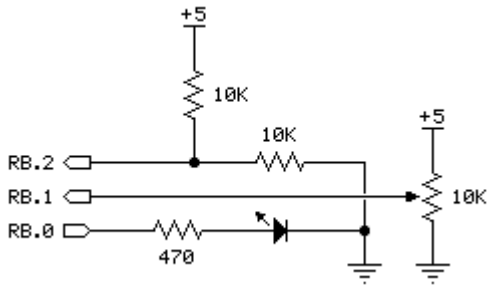
The following are points to remember with Comparator mode:

- Port B I/O pins 1 and 2 are the comparator inputs and I/O pin 0 is, optionally, the comparator result output.
- Port B I/O pin 0 may be used as a normal I/O pin by setting the OE bit of the Comparator register (**COMPARE** modes 0 and 2).
- The comparator is independent of the clock source and thus will operate even if the SX chip is halted or in SLEEP mode. To avoid spurious current draw during SLEEP mode, disable the comparator.

Related project: 8-bit ADC

COMPARE Example

Syntax




```
' ===========================================================================
'
' File...... COMPARE.SXB
' Purpose... Demonstrates the use of the SX Comparator with COMPARE
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JULY 2005
'
' ===========================================================================

' ---------------------------------------------------------------------------
' Program Description
' ---------------------------------------------------------------------------
'
' Demonstrates the use of the SX comparator with COMPARE and the auto-
' matic control of the RB.0 output state.

' ---------------------------------------------------------------------------
' Device Settings
' ---------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "COMPARE"

' ---------------------------------------------------------------------------
' Variables
' ---------------------------------------------------------------------------

result            VAR   Bit

' ===========================================================================
 PROGRAM Start
' ===========================================================================

' ---------------------------------------------------------------------------
' Program Code
' ---------------------------------------------------------------------------
Start:
  DO
    COMPARE 1, result         ' test with output to RB.0
  LOOP
```

COUNT

Example

COUNT *Pin*, *Duration*, *Variable*

Function

Count the number of cycles (0-1-0 or 1-0-1) on the specified pin during the ***Duration*** time frame and store that number in ***Variable***.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***Duration*** is a variable or constant (1 – 65535) specifying the time during which to count. The unit of time for ***Duration*** is expressed in milliseconds.
- ***Variable*** is a Byte or Word variable in which the count will be stored. Note that array elements are not allowed.

Quick Facts

	SX18 / SX20 / SX28 / SX48 / SX52
Units in *Duration*	1 ms
Duration range	1 ms to 65.535 sec
Minimum pulse width	50 / FREQ
Maximum frequency (square wave)	FREQ / 100

Explanation

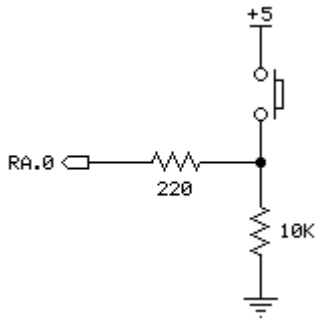
The **COUNT** instruction makes the *Pin* an input, then for the specified *Duration*, counts cycles on that pin and stores the total in *Variable*. A cycle is a change in state from 1 to 0 to 1, or from 0 to 1 to 0. Each loop in the **COUNT** routine requires 40 clock cycles, so the incoming signal must remain high or low for at least 40 cycles to be measured accurately. The maximum frequency that **COUNT** can accurately handle is about 1% the clock frequency driving the SX. A 20 MHz clock source, for example, would allow the **COUNT** instruction to count transitions in a 200 kHz square wave (or period of 5 µsecs).

If you use **COUNT** on slowly changing analog waveforms like sine waves, you may find that the value returned is higher than expected. This is because the waveform may pass through the SX's 1.4-volt logic threshold slowly enough that noise causes false counts. You can fix this by enabling the SCHMITT trigger configuration on the pin used for **COUNT**.

Related instructions: PULSIN

COUNT Example

Syntax




```
' =============================================================================
'
' File...... COUNT.SXB
' Purpose... Demonstrates SX/B COUNT instruction
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =============================================================================

' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' This program will count the transitions on the FreqIn (RA.0) pin for
' one second. Run the program in Debug mode, then select Poll from the
' Debug control panel.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "COUNT"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

FreqIn      PIN   RA.0   INPUT            ' frequency input

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Duration    CON   1000                    ' measure for one second

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

hertz       VAR   Word                    ' measured cycles
```

```
WATCH hertz                                 ' for Debug display

' ============================================================================
 PROGRAM Start
' ============================================================================

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:
   DO
     COUNT FreqIn, Duration, hertz          ' measure frequency
     BREAK                                  ' display in Watch window
   LOOP
```

DATA / WDATA

Example

Label:
DATA *Const0 {, Const1, Const2, ...}*
WDATA *Const0 {, Const1, Const2, ...}*

Function
Creates a table of data values for use with the **READ** instruction.

- ***Label*** is the symbolic name for the table and serves as a pointer to the location of table index zero.
- ***Const*** is any constant (byte, ASCII character, or computed value).

Explanation
The **DATA** and **WDATA** directives allow the programmer to create a table of [read only] values for use in the SX/B program. Using **DATA** or **WDATA** is a convenient way to store output patterns and text messages.

DATA is typically used to store byte values, **WDATA** for word values. If a value greater than 255 is used in a **DATA** table the value will be stored as two bytes, LSB first. **WDATA** always stores values as two bytes and, as above, the order is LSB, then MSB.

You must make sure the program does not attempt to execute the **DATA** or **WDATA** statements. By convention, **DATA** and **WDATA** are placed after the main program loop to prevent the execution of these statements.

```
CR    CON   13

PROGRAM Start

Start:
  LEDs = %00000000
  TRIS_LEDs = %00000000       ' make leds outputs
Main:
  idx = 0

TX_Msg:
   DO
     READ StartMsg + idx, char
     IF char = 0 THEN EXIT
     SEROUT Sio, Baud, char
     INC idx
   LOOP
   ENDIF

Show_Count:
   INC counter
   IF counter = 10 THEN
     counter = 0
   ENDIF
   READ SegMap + counter, LEDs
   PAUSE 1000
   GOTO Main
' -----------------------------------------------------------------------
SegMap:                       ' segments maps
'       .gfedcba
  DATA  %00111111             ' 0
```

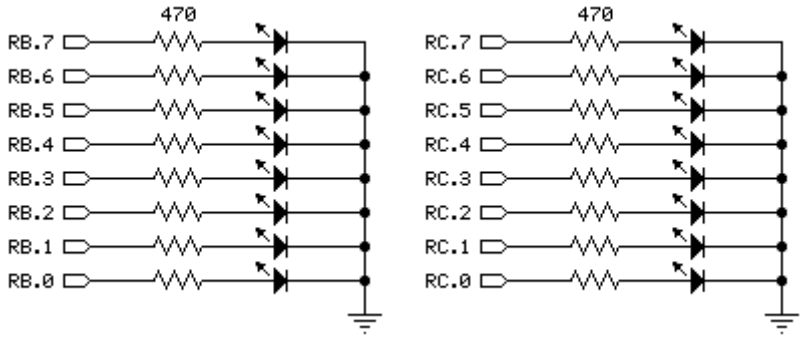
```
  DATA  %00000110             ' 1
  DATA  %01011011             ' 2
  DATA  %01001111             ' 3
  DATA  %01100110             ' 4
  DATA  %01101101             ' 5
  DATA  %01111101             ' 6
  DATA  %00000111             ' 7
  DATA  %01111111             ' 8
  DATA  %01100111             ' 9


StartMsg:
  DATA  "SX/B Really Rocks!", CR, 0

BigTable:
  WDATA %00000000_00001111
  WDATA %00000000_11110000
  WDATA %00001111_00000000
  WDATA %11110000_00000000
```

Note: When defining embedded strings as in the example above, the string may be up to 128 characters.

Related instruction: READ

DATA / WDATA Example

Syntax




```
' ================================================================================
'
' File...... WDATA.SXB
' Purpose... Display LED patterns from WDATA table
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' ================================================================================


' --------------------------------------------------------------------------------
' Program Description
' --------------------------------------------------------------------------------
'
' This program shows how to store and retrieve 16-bit values from a WDATA
' table. Note that the index for READ is updated by two for each loop
' as the WDATA table uses 2-byte values.

' --------------------------------------------------------------------------------
' Device Settings
' --------------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "WDATA"

' --------------------------------------------------------------------------------
' IO Pins
' --------------------------------------------------------------------------------

Leds        PIN   RBC   OUTPUT            ' 16-bit RB/RC port

' --------------------------------------------------------------------------------
' Variables
' --------------------------------------------------------------------------------

idx         VAR   Byte

' ================================================================================
```

```
 PROGRAM Start
' ============================================================================

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:

Main:
  DO
    READINC Chaser + idx, Leds            ' new pattern into LEDs
    PAUSE 50
    IF idx > 14 THEN                      ' end of table?
     idx = 0                              ' yes, reset
    ENDIF
  LOOP

' ============================================================================
' User Data
' ============================================================================

Chaser:
  WDATA %00000000_00001111
  WDATA %00000000_00111100
  WDATA %00000000_11110000
  WDATA %00000011_11000000
  WDATA %00001111_00000000
  WDATA %00111100_00000000
  WDATA %11110000_00000000
  WDATA %11000000_00000011
```

DEC

DEC *Variable*

Function
Decrement value of ***Variable*** by one.

- ***Variable*** is byte variable, byte array element, or word variable.

Explanation
The **DEC** instruction subtracts one from the specified variable. If the variable holds zero it will roll over to its maximum value (255 for bytes, 65535 for words) after **DEC**.

```
flags VAR Byte(2)
result VAR Word

Main:
  flags(0) = 0
  DEC flags(0)          ' flags(0) = flags(0) - 1

  result = $0000
  DEC result            ' result is now $FFFF
```

Related Instruction: INC and DJNZ

DJNZ

DJNZ *Variable, Label*

Function

Decrement value of ***Variable*** by one and jump to ***Label*** if ***Variable*** is not equal to zero.

- ***Variable*** is byte variable, byte array element, or word variable.
- ***Label*** is a program label that is followed by operational code (do not use **DJNZ** with a **DATA** or **WDATA** table label).

Explanation

The **DJNZ** instruction decrements *Variable* by one and if the result of that operation is not zero the program will jump to the location specified by *Label*.

```
Start:
  flashes = 5

Main:
  HIGH RA.0
  PAUSE 100
  LOW RA.0
  PAUSE 400
  DJNZ flashes, Main          ' flash until flashes = 0
  END
```

Related instruction: DEC

DO ... LOOP

DO { **WHILE** | **UNTIL** *Condition* }
  Statement(s)
  { **EXIT** }
LOOP

DO
  Statement(s)
  { **EXIT** }
LOOP { **NEVER** | **UNTIL** *Condition* | **WHILE** *Condition* }

Function
Create a repeating loop that executes the program lines between **DO** and **LOOP**, optionally testing before or after the loop statements.

- ***Condition*** is a simple statement, such as "idx = 7" that can be evaluated as True or False. Only one comparison operator is allowed (see IF...THEN for valid condition operators).
- ***Statement*** is any valid SX/B statement.

Explanation
DO...LOOP loops let your program execute a series of instructions indefinitely, or until a specified condition terminates the loop. The simplest form is shown here:

```
Alarm_On:
  DO
    HIGH Alarm_LED
    PAUSE 500
    LOW Alarm_LED
    PAUSE 500
  LOOP
```

In this example the alarm LED will flash until the SX is reset. **DO...LOOP** allows for condition testing before and after the loop statements as show in the examples below.

```
Alarm_On:
  DO WHILE AlarmStatus = 1
    HIGH Alarm_LED
    PAUSE 500
    LOW Alarm_LED
    PAUSE 500
  LOOP
  RETURN
```

```
Alarm_On:
  DO
    HIGH Alarm LED
    PAUSE 500
    LOW Alarm_LED
    PAUSE 500
  LOOP UNTIL AlarmStatus = 0
  RETURN
```

When the second form is used the loop statements will run at least once before the condition test.

Related instructions: FOR...NEXT and EXIT

END

END

Function
Ends program execution.

Explanation
END prevents the SX from executing any further instructions until it is reset, either externally (via MCLR\ pin) or by a watchdog timer timeout. **END** does not place the SX in low power (**SLEEP**) mode.

```
Main:
  FOR idx = 1 TO 10
  HIGH RB.0
  PAUSE 100
  LOW RB.0
  PAUSE 100
  NEXT
  END
```

Related instructions: SLEEP and RESETWDT

EXIT

{IF *Condition* THEN} **EXIT**

Function
Causes the immediate termination of a loop construct (**FOR...NEXT** or **DO...LOOP**) when ***Condition*** evaluates as True.

- ***Condition*** is a simple statement, such as "x = 7" that can be evaluated as True or False.

Explanation
The **EXIT** command allows a program to terminate a loop construct before the loop limit test is executed. For example:

```
Main:
  FOR idx = 1 TO 15
    IF idx > 9 THEN EXIT
    SEROUT TX, Baud, "*"
  NEXT
```

In this program, the **FOR idx = 1 TO 15** loop will not run past nine because the **IF idx > 9 THEN EXIT** contained within will force the loop to terminate when *idx* is greater than nine. Note that the **EXIT** command only terminates the loop that contains it. In the the program above, only nine asterisks will be transmitted on the TX pin.

Here is the **DO...LOOP** version of the same program:

```
Start:
  idx = 1

Main:
  DO
  IF idx > 9 THEN EXIT
    SEROUT TX, Baud, "*"
    INC idx
  LOOP WHILE idx <= 15
```

EXIT may also be used by itself when part of a larger **IF...THEN...ENDIF** or **DO...LOOP** block:

```
IF idx > 9 THEN
    SEROUT TX, Baud, CR
    idx = 1
    EXIT
ENDIF
```

Related Instructions: FOR...NEXT, DO...LOOP, and IF...THEN

FOR ... NEXT

Example

FOR *Variable = StartVal* **TO** *EndVal {***STEP** *{-}StepVal}*
Statement(s)
{ **EXIT** *}*
NEXT

Function

Create a repeating loop that executes the program lines between **FOR** *and* **NEXT***, incrementing or decrementing* ***ByteVar*** *according to* ***StepVal*** *until the value of* ***ByteVar*** *reaches or passes the* ***EndVal****.*

- ***Variable*** *a byte or word variable used as a loop counter.*
- ***StartVal*** *is a constant or variable that sets the starting value of the counter.*
- ***EndVal*** *is a constant or a variable that sets the ending value of the counter.*
- ***StepVal*** *is a constant or a variable by which Variable is incremented or (when negative) decremented during each iteration of the loop.*

Explanation

FOR...NEXT loops let your program execute a series of instructions for a specified number of repetitions. By default, each time through the loop, *Variable* is incremented by 1. It will continue to loop until the value of the *Variable* reaches or exceeds *EndVal*. Also, **FOR...NEXT** loops always execute at least once. The simplest form is shown here:

```
Blink_LED:
  FOR idx = 1 TO 10                ' blink 10 times
    HIGH LED                       ' light the LED
    PAUSE 200                      ' wait 0.2 secs
    LOW LED                        ' extinguish the LED
    PAUSE 300                      ' wait 0.3 secs
  NEXT
```

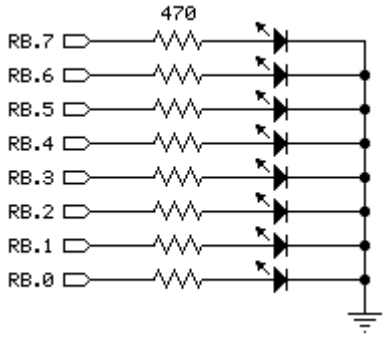
In this simple example, the **FOR** instruction initializes *idx* to 1. Then the **HIGH**, **PAUSE**, **LOW**, and **PAUSE** instructions are executed. At **NEXT**, *idx* is checked to see if it is less than 10. If it is, *idx* will be incremented by 1 and the loop instructions run again. When *idx* is equal to 10 the loop terminates and the program continues at the instruction that follows **NEXT**.

Note that when using word variables for *StartVal* or *EndVal*, *Variable* must be a word variable as well. If a *Variable* is a byte, and a word constant is used for either *StartVal* or *EndVal*, the constant value(s) will be truncated to eight bits.

Related instructions: DO...LOOP and EXIT

FOR ... NEXT Example

Syntax




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Demonstrates FOR-NEXT loops in SX/B with a "ping-pong" LED display.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "FOR-NEXT"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

LEDs        PIN   RB    OUTPUT      ' LED outputs

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

idx         VAR   Byte              ' loop counter

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:


Main:
  DO
    FOR idx = 0 TO 5                ' loop 7 times
      LEDs = 1 << idx               ' LEDs = 1, 2, 4, 8, ...
```

```
    PAUSE 100                   ' pause 100 ms
  NEXT
  FOR idx = 7 TO 1 STEP -1      ' loop 7 times
    LEDs = 1 << idx             ' LEDs = 128, 64, 32 ...
    PAUSE 100                   ' pause 100 ms
  NEXT
LOOP                            ' repeat forever
```

FREQOUT

Example

FREQOUT *Pin, Duration, Freq*

Function

Generate square wave for a specified duration.

- ***Pin*** is any SX IO pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***Duration*** is a byte variable/constant (1 - 255) specifying the amount of time to generate the tone. The unit of time for ***Duration*** is 1 millisecond.
- ***Freq*** is a constant (1 - 65535) specifying the frequency of the square wave.

Quick Facts

	Low Limit	High† Limit
Frequency	FREQ ÷ 500,000	FREQ ÷ 32

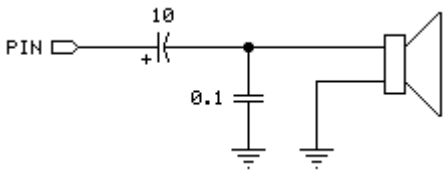
† **Note:** For the greatest possible accuracy, limit high frequency value to FREQ ÷ 512.

Explanation

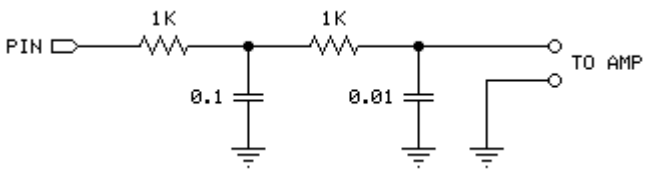
FREQOUT generates a square wave on an I/O pin. The output pin may be connected as shown in the circuits below for audio use. Other applications include IR LED modulation.

Note that the *Duration* can be affected by *Freq* parameter. For example, a frequency of 100 Hz has a period of 10 milliseconds, hence the shortest possible *Duration* value is 10. Even at this minimum, only one cycle would be produced and this may not be practical or useful.

When driving a Hi-Z speaker (> 40 Ω) or piezo element:



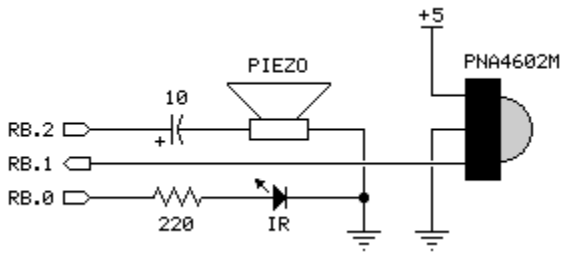
When connecting to an audio amplifier:



Related instruction: SOUND

FREQOUT Example

Syntax




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Uses FREQOUT to modulate an IR LED for object detection. When object
' is detected, a tone is generated from the piezo element.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "FREQOUT"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

IrLED             PIN   RB.0  OUTPUT      ' output to IR LED
Detect            PIN   RB.1  INPUT ' input from IR detector
Spkr        PIN   RB.2  OUTPUT      ' output to piezo

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

IrMod             CON   38000                   ' modulation freq = 38 kHz
Yes         CON   0                 ' for active-low output
No          CON   1

' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  DO
    FREQOUT IrLED, 1, IrMod ' modulate IR diode
    IF Detect = Yes THEN ' check detector
      SOUND Spkr, 100, 2 ' buzz if object detected
    ENDIF
```

```
LOOP
```

GET

GET *Location, Variable {, Variable, ... }*

Function
Copy value(s) from RAM into ***Variables(s)***, starting at ***Location***.

- ***Location*** is the starting address to be copied.
- ***Variable*** is byte variable, byte array element, or word variable.

Explanation
The **GET** instruction provides a convenient method for copying multiple consecutive values from RAM into individual byte variables. For example,

```
  hrs = clock(0)
  mins = clock(1)
  secs = clock(2)
```

... can be simplified to a single line of code:

```
GET clock(0), hrs, mins, secs
```

Note that this works because array elements are stored as addresses.

When *Variable* is a word, its LSB is read from *Location* and its MSB from *Location* + 1. For example:

```
loVal             VAR   Byte
hiVal             VAR   Byte
result            VAR   Word

Start:
  loVal = $33
  hiVal = $CC
  GET @loVal, result                    ' result = $CC33
```

Using GET with Subroutine Parameters

When a simple variable is passed to a subroutine by address (using @), as in:

```
GOSUB Some_Routine, @aValue ' pass address of 'aValue'
```

... **GET** can be used to retrieve the value from that address within the subroutine:

```
Some_Routine:
  rtnAddr = __PARAM1                    ' save return address
  GET rtnAddr, theValue                 ' get value from address
  ...                                   ' do something with the value
  PUT rtnAddr, newValue                 ' update value at passed address
  RETURN
```

This technique allows the subroutine to accept (and potentially modify) any defined variable. Note that for single-parameter instances, the __RAM() system array may be used in place of **GET** and **PUT**:

```
Some_Routine:
  rtnAddr = __PARAM1                    ' save return address
  theValue = __RAM(rtnAddr)             ' get value from address
  ...                                   ' do something with the value
  __RAM(rtnAddr) = newValue             ' update value at passed address
  RETURN
```

Note: As of SX/B 1.2, Subroutines (and now Functions in SX/B 1.5) can return a value directly so passing the address of a variable (with @) is not required unless multiple variable addresses are to be passed to the Subroutine/Function.

Related instruction: PUT

GOSUB ... RETURN

Examples

GOSUB *Label* . . . **RETURN** *{ Value }*

Function

Store the address of the next instruction after **GOSUB**, then go to the point in the program specified by ***Label***; with the intention of returning to the stored address.

- ***Label*** specifies the name of the subroutine to run.
- ***Value*** is an optional value/variable to return

Quick Facts

	SX18, SX20, SX28, SX48, SX52
Maximum Nested **GOSUB**s	8

Explanation

GOSUB is a close relative of **GOTO**, in fact, its name means, "GO to a SUBroutine". When a program reaches a **GOSUB**, the program executes the code beginning at the specified *Label*. Unlike **GOTO**, **GOSUB** also stores the address of the instruction immediately following itself. When the program encounters the **RETURN** instruction, it interprets it to mean, "go to the instruction that follows the most recent GOSUB." In other words, a **GOSUB** makes the program do a similar operation as you do when you see a table or figure reference in this manual; 1) you remember where you are, 2) you go to the table or figure and read the information there, and 3) when you've reached the end of it, you "return" to the place you were reading originally.

GOSUB is mainly used to execute the same piece of code from multiple locations. If you have, for example, a block of three lines of code that need to be run from 10 different locations in your entire program you could simple copy and paste those three lines to each of those 10 locations. This would amount to a total of 30 lines of repetitive code (and extra space wasted in the program memory). A better solution is to place those three lines in a separate routine, complete with it's own label and followed by a **RETURN** instruction, then just use a **GOSUB** instruction at each of the 10 locations to access it. Since SX/B compiles instructions inline (no optimization) this technique can save a lot of program space.

SX/B simplifies subroutine use and error trapping with the declaration of subroutines (**SUB** directive) and required/possible parameters. When a subroutine is declared, the **GOSUB** keyword is no longer required and any parameters passed with be checked against the user declaration. The following examples demonstrate the differences in code style.

Version 1.1 (This style is still valid but not recommended)

```
Get_Char:
  SERIN Sio, Baud, char                    ' wait for character
  RETURN

' ------------------------------

Start:
  TRIS_B = %00000000                       ' make RB pins outputs

Main:
  GOSUB Get_Char
  IF char <> "!" THEN Main                 ' wait for "!"
```

Version 1.2+ (applies to **SUB** and **FUNC**)

```
GET_CHAR SUB                                ' subroutine (no parameters)

' ------------------------------

Start:
  TRIS_B = %00000000                        ' make RB pins outputs

Main:
  char = GET_CHAR                           ' no "GOSUB" required
  IF char <> "!" THEN Main                  ' wait for "!"
  END

' ------------------------------

SUB GET_CHAR
  SERIN Sio, Baud, temp1                    ' wait for character
  RETURN temp1                              ' return character to caller
ENDSUB
```

Declared subroutines simplify SX/B programming by removing the necessity of the **GOSUB** keyword (which, in effect, allows the programmer to extend the language by creating new commands), it allows the compiler to validate the number of parameters being passed, and -- most valuable to the programmer -- it allows subroutine code to be placed anywhere in memory without concern of code page boundaries (now handled automatically).

Passing Parameters To/From a Subroutine

SX/B allows the programmer to pass up to four parameters to subroutines. The parameter may hold a value (bit or byte) or the address of a byte-variable (when prefaced with '@'). When used in subroutines, passed parameters must be saved before any SX/B instructions are called.

For example:

```
TX_BYTE     SUB                     ' subroutine with no parameters
SEND_CHAR   SUB   2                 ' subroutine with 2 parameters

' -----------------------------

Main:
  ' less convenient
  theChar = "*"                     ' byte to send
  idx = 10                          ' times to send
  TXBYTE
  theChar = 13
  idx = 1
  TXBYTE

' much more convenient
  SENDCHAR "*", 10                      ' **********
  SENDCHAR 13, 1                        ' <CR>
  PAUSE 1000
  GOTO Main

' -----------------------------

SUB SEND_CHAR
  theChar = __PARAM1                    ' save character to send
  idx = __PARAM2                    ' times to send character

  TX_BYTE
ENDSUB

SUB TX BYTE
  DO WHILE idx > 0
    SEROUT Sio, Baud, theChar                ' send the character
    DEC idx                             ' update count, exit if 0
  LOOP
ENDSUB
```

This subroutine (SEND_CHAR) expects two parameters: the character to transmit (using **SEROUT**), and the number of times to send the character.

A subroutine can be constructed to modify any variable that is passed to it (by address using '@'). For example:

```
INVERT_BITS       SUB   1               ' subroutine with 1 parameter

' -----------------------------

Start:
  TRIS_B = %00000000                    ' make RB pins outputs

Main:
```

```
  myBits = $A5                          ' myBits = %10100101
  INVERT_BITS @myBits ' pass address of variable
  RB = myBits ' RB = %01011010 ($5A)
  END

' ------------------------------

SUB INVERT_BITS
  regAddr = __PARAM1                    ' save address
  GET rtnAddr, regVal                   ' get value from address
  regVal = ~regVal                      ' invert bits
  PUT rtnAddr, regVal                   ' update passed variable
ENDSUB
```

An easier method, however, is to allow the subroutine to pass a value directly back to the caller. This update to the program above performs the same function, yet is easier to understand and prevents possible errors resulting is missing '@' headers.

```
INVERT_BITS SUB 1                       ' subroutine with 1 parameter

' ------------------------------

Start:
 TRIS_B = %00000000                     ' make RB pins outputs
Main:
 myBits = $A5                           ' myBits = %10100101
 myBits = INVERT_BITS myBits            ' pass value, get one back
 RB = myBits                            ' RB = %01011010 ($5A)
 END

' ------------------------------

SUB INVERT_BITS
  regVal = __PARAM1                     ' get value from caller
  regVal = ~regVal                      ' invert bits
  RETURN regVal                    ' return value to caller
ENDSUB
```

Notice that this style eliminates the need for a variable that holds the address of the target variable and simplifies the subroutine code. By using a defined function (with **FUNC**) the subroutine can return a word value.

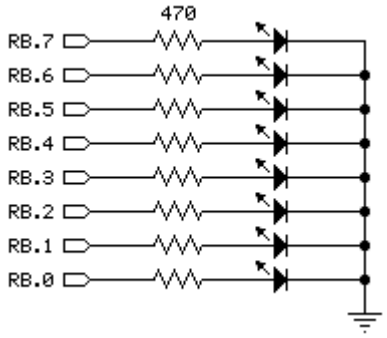
Passing Strings

SX/B allows the programmer to pass a literal or stored (with **DATA**) string to a subroutine. String passing requires at least two parameters to handle the base and offset address bytes to the string (these values are used by **READ**). See READ for an example of string use in SX/B.

related instruction: GOTO

GOSUB...RETURN Example

Syntax




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Demonstrates GOSUB with parameter passing across page boundaries. Note
' that the GOSUB keyword is no longer required when using version 1.2
' syntax (SUB definition). Note, too, that the INVERT8 function returns
' an 8-bit value even when a 16-bit (word) is passed as a parameter.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "GOSUB"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

LEDs        PIN   RB    OUTPUT


' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

Speed             CON   100               ' loop delay control

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

zigzag            VAR   Byte              ' zigzag controller
tmpW1             VAR   Word              ' subroutine work var

' =======================================================================
 PROGRAM Start
' =======================================================================
```

```
' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

DELAY             SUB   1, 2              ' delay in milliseconds
INVERT8     FUNC  1, 1, 2           ' invert bits in byte
INVERT16    FUNC  2, 1, 2           ' invert bits in word

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:


Main:
  zigzag = %00000011
  DO
    LEDs = INVERT8 zigzag                ' invert LED pattern
    DELAY Speed                     ' loop delay
    zigzag = zigzag << 1                 ' shift bit left
  LOOP UNTIL zigzag = %10000000
  zigzag = %10000000
  DO
    LEDs = INVERT8 zigzag
    DELAY Speed
    zigzag = zigzag >> 1                 ' shift bit right
  LOOP UNTIL zigzag = %00000001
  GOTO Main

' -----------------------------------------------------------------------
' Subroutines Code
' -----------------------------------------------------------------------

' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535

DELAY:
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                     ' save byte value
  ELSE
    tmpW1 = __WPARAM12                   ' save word value
  ENDIF
  PAUSE tmpW1
  RETURN

' -----------------------------------------------------------------------

' Use: aVar = INVERT theByte
' -- inverts the bits in 'theByte'
' -- returns 8-bit value, even if 'theByte' is a word (returns inverted LSB)

FUNC INVERT8
  tmpW1_LSB = __PARAM1                   ' get current value
  tmpW1_LSB = ~tmpW1_LSB                 ' invert the bits
  RETURN tmpW1_LSB                       ' return byte to caller
ENDFUNC

' -----------------------------------------------------------------------
```

```
' Use: aVar = INVERT theWord
' -- inverts the bits in 'theWord'
' -- returns 16-bit value, even if 'theWord' is a byte

FUNC INVERT16
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                    ' convert byte to word
  ELSE
    tmpW1 = __WPARAM12                  ' save word value
  ENDIF
    tmpW1 = ~tmpW1                      ' invert the bits
  RETURN tmpW1                          ' return word to caller
ENDFUNC
```

The following example demonstrates the ability to return more than two bytes from a function:

```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Demonstrates the use of a function and a method for collecting all
' returned bytes when simple (non-array) variables are used.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "FUNC"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------
' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------
' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

result            VAR   Word              ' 32-bit result
resultHi    VAR   Word
bigVal            VAR   Byte(4)
tmpW1             VAR   Word              ' subroutine work vars
tmpW2             VAR   Word
tmpW3             VAR   Word
WATCH result, 32, UHEX                    ' display 32-bit result
WATCH bigVal, 32, UHEX

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

MULT32            FUNC  4, 2, 4
```

```
BREAK_NOW   SUB 0

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  result = MULT32 $FFFF, $0100                  ' get low word
  resultHi = __PARAM3, __PARAM4           ' get high word
  BREAK_NOW
  bigVal = MULT32 $1234, $10              ' all return bytes assigned
  BREAK_NOW
  END

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' Use: MULT32 value1, value2
' -- multiplies two values
' -- when mixing a word and byte, the word must be declared first
FUNC MULT32
  IF __PARAMCNT = 2 THEN                  ' byte * byte
    tmpW1 = __PARAM1
    tmpW2 = __PARAM2
   ENDIF
  IF __PARAMCNT = 3 THEN                  ' word * byte
    tmpW1 = __WPARAM12
    tmpW2 = __PARAM3
  ENDIF
  IF __PARAMCNT = 4 THEN                  ' word * word
    tmpW1 = __WPARAM12
    tmpW2 = __WPARAM34
  ENDIF
  tmpW3 = tmpW1 ** tmpW2                  ' calculate high word
  tmpW2 = tmpW1 * tmpW2                   ' calculate low word
  RETURN tmpW2, tmpW3                     ' return 32 bits, LSW first
ENDFUNC

' -----------------------------------------------------------------------

' Allows multiple breakpoints in program.

SUB BREAK_NOW
  BREAK
ENDSUB
```

GOTO

GOTO *Label*

Function
Jump to the point in the program specified by *Label*.

- ***Label*** is a program label that is followed by operational code (do not use **GOTO** with a **DATA** or **WDATA** table label).

Explanation
The **GOTO** instruction forces the SX to jump to the line of code that immediately follows *Label*. A common use for **GOTO** is to create endless loops; programs that repeat a group of instructions over and over. For example:

```
Main:
  INC LEDs                    ' update count
  PAUSE 100                        ' delay 0.1 seconds
  GOTO Main                        ' keep going
```

Related instructions: IF ... THEN and GOSUB

HIGH

Example

HIGH *Pin*

Function
Make the specified *Pin* an output and high (1).

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).

Explanation
The **HIGH** instruction makes the specified *Pin* an output, and then sets its value to 1 (Vdd). For example:

```
HIGH RA.3
```

Does the same thing as:

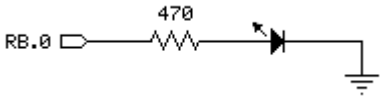
```
OUTPUT RA.3
RA.3 = 1
```

Using the **HIGH** instruction is more convenient in this case

HIGH can also be used on a whole port. For example, **HIGH RB** will make all pins on port RB high.

Related instructions: LOW, OUTPUT, and TOGGLE

HIGH / LOW Example




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Simple LED blinker using HIGH and LOW.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "HIGH-LOW"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

LED         PIN   RB.0  OUTPUT      ' LED pin

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

OnDelay     CON   150         ' time LED is on
OffDelay    CON   350         ' time LED is off

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  DO
    HIGH LED                        ' turn LED on
    PAUSE OnDelay                   ' delay
    LOW LED                         ' turn LED off
    PAUSE OffDelay                  ' delay
  LOOP                              ' repeat forever
```

I2CRECV

Example

I2CRECV *Pin*, *ByteVar*, *AckVal*

Function

Receives ***ByteVar*** from the I2C bus defined by ***Pin***.

- ***Pin*** defines the SDA pin of the I2C bus. *Pin* may any I/O pin except RA.3 (RA.7 on the SX52), RB.7, RC.7, RD.6, or RE.6 (see below).
- ***ByteVar*** is a byte variable that will hold the value returned by the slave device.
- ***AckVal*** is a bit-value that will be sent to the slave after *ByteVar* has been received (Ack = 0, Nak = 1).

Quick Facts

	SX18/20	SX28	SX48	SX52
SDA pin	RA.0 .. RA.2 RB.0 .. RB.6	RA.0 .. RA.2 RB.0 .. RB.6 RC.0 .. RC.6	RA.0 .. RA.2 RB.0 .. RB.6 RC.0 .. RC.6 RD.0 .. RD.6 RE.0 .. RE.6	RA.0 .. RA.6 RB.0 .. RB.6 RC.0 .. RC.6 RD.0 .. RD.6 RE.0 .. RE.6
SCL pin	Next pin in same group as SDA assignment.			
Transmission rate	Approximately 50 kBits/sec.			
Special Notes	Both the SDA and SCL pins must have 4.7 kΩ pull-up resisters.			

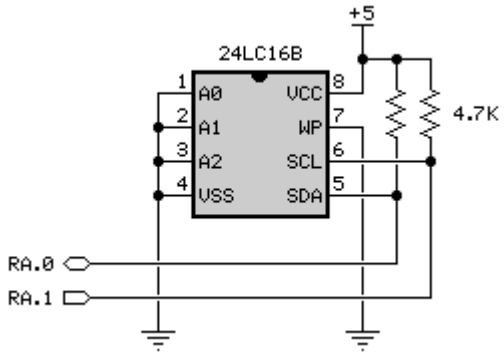
Explanation

The I2C protocol is a form of synchronous serial communication developed by Philips Semiconductor. It only requires two IO pins and both pins can be shared between multiple I2C devices. The **I2CRECV** instruction reads an eight-bit value from a previously addressed device on the I2C bus (SDA and SCL pins).

Note that the SCL pin is automatically assigned to the next higher pin in the same group as the SDA pin, so the SDA pin may *not* be assigned to RA.3 (RA.7 on the SX52), RB.7, RC.7, RD.7, or RE.7.

Related instructions: I2CSEND, I2CSTART, and I2CSTOP

I2C Example




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Writes a pseudo-random value to a 24LC16B EEPROM and then reads the
' value back. Run program in Debug/Poll mode to view address, output and
' input values.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "I2C"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

SDA         PIN   RA.0 INPUT PULLUP
SCL         PIN   RA.1 INPUT PULLUP

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

SlaveID     CON   $A0              ' for 24LC16B
Ack         CON   0
Nak CON 1

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

addr        VAR   Word              ' address in 24LC16B
addrLo            VAR   addr_LSB
addrHi            VAR   addr_MSB

outVal            VAR   Byte              ' to 24LC16B
inVal             VAR   Byte              ' from 24LC16B
```

```
tmpW1             VAR   Word              ' work vars
tmpB1             VAR   Byte
tmpB2             VAR   Byte
tmpB3             VAR   Byte
tmpB4             VAR   Byte

WATCH addr                                ' for Debug/Poll mode
WATCH outVal
WATCH inVal

' ============================================================================
 PROGRAM Start
' ============================================================================

' ----------------------------------------------------------------------------
' Subroutine Declarations
' ----------------------------------------------------------------------------

MEM_OUT     SUB   3                       ' write value to memory
MEM_IN            FUNC  1,2               ' read byte from memory

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:


Main:
  DO
    FOR addr = 0 TO $03FF
      RANDOM outVal ' recreate new value
      MEM_OUT addr, outVal ' send to 24LC16
      PAUSE 500 ' delay for Debug/Poll mode
      inVal = MEM_IN addr ' get from 24LC16
       BREAK ' allow WATCH window view
    NEXT
  LOOP

' ----------------------------------------------------------------------------
' Subroutines Code
' ----------------------------------------------------------------------------

' Use: MEM_OUT address, value
' -- writes 'value' to 24LC16B location at 'address'

SUB MEM_OUT
  tmpW1 = __WPARAM12                      ' copy address
  tmpB1 = __PARAM3                        ' copy value
  I2CSTART SDA
  tmpW1_MSB = tmpW1_MSB & $03             ' get block value
  tmpW1_MSB = tmpW1_MSB << 1
  tmpW1_MSB = tmpW1_MSB | SlaveID         ' create control byte
  tmpW1_MSB.0 = 0                         ' set RW bit for write
  I2CSEND SDA, tmpW1_MSB                  ' send slave ID
  I2CSEND SDA, tmpW1_LSB                  ' send word address
  I2CSEND SDA, tmpB1                      ' send data byte
  I2CSTOP SDA                             ' finish
  ENDSUB
```

```
' ------------------------------------------------------------------------

' Use: value = MEM_IN address
' -- reads 'value' from 24LC16B location at 'address'

FUNC MEM_IN
  tmpW1 = __WPARAM12                    ' copy address
  I2CSTART SDA
  tmpW1_MSB = tmpW1_MSB & $03           ' get block value
  tmpW1_MSB = tmpW1_MSB << 1
  tmpW1_MSB = tmpW1_MSB | SlaveID       ' create control byte
  tmpW1_MSB.0 = 0                       ' set RW bit for write
  I2CSEND SDA, tmpW1_MSB                ' send slave ID
  I2CSEND SDA, tmpW1_LSB                ' send word address
  I2CSTART SDA                          ' restart for read
  tmpW1_MSB.0 = 1                       ' set RW bit for Read
  I2CSEND SDA, tmpW1_MSB                ' resend slave ID
  I2CRECV SDA, tmpB1, Nak               ' get one byte
  I2CSTOP SDA
  RETURN tmpB1
ENDFUNC
```

I2CSEND

Example

I2CSEND *Pin*, *ByteVal* {, *AckVar*}

Function

Sends ***ByteVal*** on the I2C bus defined by ***Pin***.

- ***Pin*** defines the SDA pin of the I2C bus. *Pin* may any I/O pin except RA.3 (RA.7 on the SX52), RB.7, RC.7, RD.6, or RE.6 (see below).
- ***ByteVal*** is a variable or constant (0 - 255) that will be transmitted on the I2C bus.
- ***AckVar*** is an optional bit-variable that will hold the Ack/Nak status bit returned by the slave device.

Quick Facts

	SX18/20	SX28	SX48	SX52
SDA pin	RA.0 .. RA.2 RB.0 .. RB.6	RA.0 .. RA.2 RB.0 .. RB.6 RC.0 .. RC.6	RA.0 .. RA.2 RB.0 .. RB.6 RC.0 .. RC.6 RD.0 .. RD.6 RE.0 .. RE.6	RA.0 .. RA.6 RB.0 .. RB.6 RC.0 .. RC.6 RD.0 .. RD.6 RE.0 .. RE.6
SCL pin	Next pin in same group as SDA assignment.			
Transmission rate	Approximately 50 kBits/sec.			
Special Notes	Both the SDA and SCL pins must have 4.7 kΩ pull-up resisters.			

Explanation

The I2C protocol is a form of synchronous serial communication developed by Philips Semiconductor. It only requires two I/O pins and both pins can be shared between multiple I2C devices. The **I2CSEND** instruction transmits an eight-bit value to a previously addressed device on the I2C bus (SDA and SCL pins).

Note that the SCL pin is automatically assigned to the next higher pin in the same group as the SDA pin, so the SDA pin may *not* be assigned to RA.3 (RA.7 on the SX52), RB.7, RC.7, RD.7, or RE.7.

Related instructions: I2CRECV, I2CSTART, and I2CSTOP

I2CSTART

Example

I2CSTART *Pin*

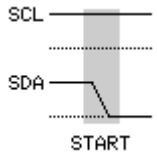
Function
Generates an I2C Start condition.

- ***Pin*** defines the SDA pin of the I2C bus. *Pin* may any I/O pin except RA.3 (RA.7 on the SX52), RB.7, RC.7, RD.6, or RE.6 (see below).

Quick Facts

	SX18/20	SX28	SX48	SX52
SDA pin	RA.0 .. RA.2 RB.0 .. RB.6	RA.0 .. RA.2 RB.0 .. RB.6 RC.0 .. RC.6	RA.0 .. RA.2 RB.0 .. RB.6 RC.0 .. RC.6 RD.0 .. RD.6 RE.0 .. RE.6	RA.0 .. RA.6 RB.0 .. RB.6 RC.0 .. RC.6 RD.0 .. RD.6 RE.0 .. RE.6
SCL pin	Next pin in same group as SDA assignment.			
Transmission rate	Approximately 50 kBits/sec.			
Special Notes	Both the SDA and SCL pins must have 4.7 kΩ pull-up resisters.			

Explanation
The I2C protocol is a form of synchronous serial communication developed by Philips Semiconductor. It only requires two I/O pins and both pins can be shared between multiple I2C devices. The **I2CSTART** instruction generates an I2C Start condition on the I2C bus (SDA and SCL pins); this condition is used to start (or restart) a transmission sequence.



Note that the SCL pin is automatically assigned to the next higher pin in the same group as the SDA pin, so the SDA pin may *not* be assigned to RA.3 (RA.6 on the SX52), RB.7, RC.7, RD.7, or RE.7.

Special Note
The **I2CSTART** instruction monitors the state of the SCL line and will wait for SCL to be high before returning; this could cause some systems to hang if the SCL line is shorted to Vss. To prevent this hang, the programmer can make the SCL line an input then test its state before calling **I2CSTART**.

Related instructions: I2CSTOP, I2CSEND, and I2CRECV

I2CSTOP

Example

I2CSTOP *Pin*

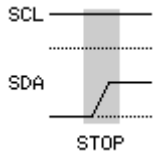
Function
Generates an I2C Stop condition.

- ***Pin*** defines the SDA pin of the I2C bus. *Pin* may any I/O pin except RA.3 (RA.7 on the SX52), RB.7, RC.7, RD.6, or RE.6 (see below).

Quick Facts

	SX18/20	SX28	SX48	SX52
SDA pin	RA.0 .. RA.2 RB.0 .. RB.6	RA.0 .. RA.2 RB.0 .. RB.6 RC.0 .. RC.6	RA.0 .. RA.2 RB.0 .. RB.6 RC.0 .. RC.6 RD.0 .. RD.6 RE.0 .. RE.6	RA.0 .. RA.6 RB.0 .. RB.6 RC.0 .. RC.6 RD.0 .. RD.6 RE.0 .. RE.6
SCL pin	Next pin in same group as SDA assignment.			
Transmission rate	Approximately 50 kBits/sec.			
Special Notes	Both the SDA and SCL pins must have 4.7 kΩ pull-up resisters.			

Explanation
The I2C protocol is a form of synchronous serial communication developed by Philips Semiconductor. It only requires two I/O pins and both pins can be shared between multiple I2C devices. The **I2CSTOP** instruction generates an I2C Stop condition on the I2C bus (SDA and SCL pins); this condition is used to terminate a transmission sequence.



Note that the SCL pin is automatically assigned to the next higher pin in the same group as the SDA pin, so the SDA pin may *not* be assigned to RA.3 (RA.6 on the SX52), RB.7, RC.7, RD.7, or RE.7.

Related instructions: I2CSTART, I2CSEND, and I2CRECV

IF ... THEN ... ELSE ... ENDIF

IF *Condition* **THEN**
statement(s)
{ **ELSE** | **ELSEIF**
statement(s) }
ENDIF

Function

Evaluate *Condition* and, if it is true, run the code block that follows **THEN**, otherwise jump to the (optional) code block that follows **ELSE**. If no **ELSE** block is provided, the program will continue at the line that follows **ENDIF**.

- ***Condition*** is a simple statement, such as "x = 7" that can be evaluated as True or False.
- ***Statement*** is any valid SX/B program statement.

Explanation

IF...THEN...ELSE is a primary decision maker that allows one block of code or [optionally] another to run based on the result (True or False) of a condition. The available comparison operators are:

Comparison Operator	Definition
=	Equal
<>	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

Comparisons are always written in the form: *Variable Op Value*.

- *Variable* is a bit, byte or word variable.
- *Op* is the comparison operator.
- *Value* is a variable or constant for comparison.

This simple example shows how **IF...THEN...ELSE** is used with a subroutine that can accept a byte or word parameter.

```
' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                    ' save byte value
  ELSE
    tmpW1 = __WPARAM12                  ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB
```

Related instruction: IF ... THEN

IF ... THEN

IF *Condition* [**THEN** | **GOTO**] [*Label* | **EXIT**]

Function
Evaluate *Condition* and, if it is true, jump to the point in the program designated by *Label*.

- ***Condition*** is a simple statement, such as "x = 7" that can be evaluated as True or False.
- ***Label*** specifies where to go in the event that *Condition* statement evaluates as True.

Explanation
IF...THEN is decision maker that affects program flow. It tests a *Condition* statement and, if that statement is True, goes to a point in the program specified by *Label*. The available comparison operators are:

Comparison Operator	Definition
=	Equal
<>	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

Comparisons are always written in the form: *Variable Op Value*.

- *Variable* is a bit, byte or word variable.
- *Op* is the comparison operator.
- *Value* is a variable or constant for comparison.

```
Main:
  IF StartBtn = 1 THEN Main                ' wait for input to go low


Check_Mode:
  IF ModePin = 0 GOTO Show_AMPM            ' use HH:MM xM format if mode = 0
```

Some programmers may prefer a more verbose style; the following syntax is also supported:

```
Check_Mode:
  IF ModePin = 0 THEN GOTO Show_AMPM             ' use HH:MM xM format if mode = 0
```

Related instructions: IF ... THEN ... ELSE, BRANCH, GOTO, and EXIT

INC

INC *Variable*

Function
Increment value of *Variable* by one.

- ***Variable*** is byte variable, byte array element, or word variable.

Explanation
The **INC** instruction adds one from the specified variable. If the variable holds its maximum value (255 for bytes, 65535 for words), it will roll over to zero after **INC**.

```
flags             VAR   Byte(2)
result            VAR   Word

Main:
  flags(0) = 0
  INC flags(0)                ' flags(0) = flags(0) + 1
  result = $FFFF
  INC result                  ' result is now $0000
```

Related Instruction: DEC

INPUT

INPUT *Pin*

Function
Make the specified *Pin* an input by writing a one (1) to the corresponding bit of the associated port TRIS register.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).

Explanation
There are several ways to make a pin an input. When an SX/B program is reset, all of the IO pins are made inputs. Instructions that rely on input pins, like **PULSIN** and **SERIN**, automatically change the specified pin to input mode. Writing 1s to particular bits of the port TRIS register makes the corresponding pins inputs. And then there's the **INPUT** instruction.

```
Start:
  INPUT RA.3

Hold:
  IF RA.3 = 1 THEN Hold             ' stay until RA.3 = 0
```

What happens if your program writes to a port bit of a pin that is set up as an input? The value is stored in the port register, but has no effect on the outside world. If the pin is changed to output, the last value written to the corresponding port bit will appear on the pin

INPUT can also be used on a whole port; for example **INPUT RB** will make all pins on port RB inputs.

Related instructions: OUTPUT and REVERSE

INTERRUPT ... RETURNINT

Examples

INTERRUPT {NOPRESERVE | NOCODE} {Rate} *Instruction(s)* ... **RETURNINT** *{Cycles}*

Function

INTERRUPT allows the insertion of a code block to handle an interrupt in an SX/B program. The interrupt code block is terminated with **RETURNINT**.

- **NOPRESERVE** is an optional command the eliminates the automatic save/restore of the __PARAMx and M registers.
- **NOCODE** is an optional command that, when used, forces the ISR to set the FSR to zero to access variables in location $10 to $1F reliably.
- ***Rate*** is an optional constant that allows the ISR to automatically be set to a specific rate, specified in *calls per second*. *Rate* must be specified after the NOPRESERVE or NOCODE parameter if used.
- ***Cycles*** is an optional byte-variable or constant (0 - 255) that sets the number of RTCC instructions for a periodic interrupt. It is not necessary to specify *Cycles* if the *Rate*parameter is used.

Explanation

An interrupt handler allows the SX to perform "background" task, at regular intervals (using the internal RTCC rollover) or asynchronously (using the external RTCC input pin, or Port B inputs). The interrupt code must be located in Page 0 of your program, and before any other code -- this is a requirement of the SX microcontroller.

SX/B automatically saves internal program variables (__PARAMx) at the start of the ISR and restores them on exit of the ISR. This consumes ISR cycles (14 cycles on entry, 13 cycles on exit) and must be accounted for in time-critical ISR applications. If high-level instructions will not be used in the ISR, the **NOPRESERVE** option may be specified, eliminating the code that preserves and restores internal SX/B program variables. If the **NOPRESERVE**option is used, you must make sure that none of the __PARAMx variables are modified by the ISR (i.e., use high-level SX/B instructions). If the **NOCODE** option is specified then the interrupt routine must set FSR to zero to access variables in location $10 to $1F reliably.

When the interrupt *Rate* is specified, in calls per second, the SX/B compiler will automatically generate the proper !OPTION register and RETURNINT values in the initialization code. If a rate is specified that is not possible within specified FREQ setting, the Rate parameter will flag an "Invalid Parameter" error.

Periodic Interrupts

The SX chip can be set to cause an interrupt upon rollover of the Real Time Clock Counter (RTCC). By configuring an interrupt on RTCC rollover, the SX chip can perform an operation at a predefined time interval in a deterministic fashion. This can be configured by setting the STACKX or OPTIONX fuse (in the DEVICE directive, and required by SX/B) and writing to the RTI, RTS and RTE bits of the Option register (OPTION). The RTCC rollover interrupt is disabled by default.

To configure the RTCC rollover interrupt:

1. Set the STACKX or OPTIONX fuse in the DEVICE directive.
2. Write to the RTI, RTS and RTE bits of the OPTION register to enable RTCC interrupts. For RTI, a high bit (1) disables RTCC rollover interrupts and a low bit (0) enables RTCC rollover interrupts. For RTS, a high bit (1) selects incrementing RTCC on internal clock cycle and a low bit (0) increments RTCC on the RTCC pin transitions. For RTE, a high bit (1) selects incrementing on low-to-high transition and a low bit (0) increments on a high-to-low transition.

An interrupt handler that uses the RTCC rollover to create a periodic interrupt might look as follows:

```
' -----------------------------------------------------------------------
  INTERRUPT
' -----------------------------------------------------------------------
ISR_Start:
  INC tix                     ' update tix counter
  IF tix = 200 THEN                 ' check for 1 second
    tix = 0
    INC secs
    IF secs = 60 THEN
      secs = 0
      INC mins
      IF mins = 60 THEN
        mins = 0
        INC hrs
        IF hrs = MaxHr THEN
          hrs = 0
        ENDIF
      ENDIF
    ENDIF
  ENDIF

ISR_Exit:
  RETURNINT 156
```

This routine (ISR) maintains a real-time clock. The variable called *tix* is updated each time through the ISR and when it reaches 200 the seconds counter is updated. This indicates that the ISR is designed to run 200 times per second.

Even when using a 4 MHz oscillator, each instruction takes only 0.25 microseconds. By using the RTCC prescaler we can slow the RTCC timing to a more manageable value and simplify the code. For the timer above, the RTCC prescaler ratio is set to 1:128 via the OPTION register (and clearing OPTION.6 enables the ISR):

```
OPTION = $86 ' prescaler = 1:128
```

Note that the *Cycles* count is set to 156 -- this means that the ISR will run after 156 cycles of the internal RTCC. The final math for the ISR timing works out like this:

4 MHz (FREQ) ÷ 128 (prescaler) ÷ 200 (ISR rate) = 156.25 (*Cycles*)

Option Register

The OPTION register is a run-time writable register used to configure the RTCC and the Watchdog Timer. The size of this register is affected by the OPTIONX device setting.

OPTION							
7	6	5	4	3	2	1	0
RTW	RTI	RTS	RTE	PSA	PS2	PS1	PS0

When OPTION Extend = 0, bits 7 and 6 are implemented.
When OPTION Extend = 1, bits 7 and 6 read as '1's.

RTW If = 0, register $01 is W
If = 1, register $01 is RTCC

RTI If = 0, RTCC roll-over interrupt is enabled
If = 1, RTCC roll-over interrupt is disabled

RTS If = 0, RTCC increments on internal instruction cycle
If = 1, RTCC increments on transition of RTCC pin

RTE If = 0, RTCC increments on low-to-high transition
If = 1, RTCC increments on high-to-low transition

PSA If = 0, prescaler is assigned to RTCC, divide rate determined by PS2..PS0 bits
If = 1, prescaler is assigned to WDT, and divide rate on RTCC is 1:1

Prescaler Division Ratios		
PS2, PS1, PS0	RTCC Divide Rate	Watchdog Timer Divide Rate
000	1:2	1:1
001	1:4	1:2
010	1:8	1:4
011	1:16	1:8
100	1:32	1:16
101	1:64	1:32
110	1:128	1:64
111	1:256	1:128

Using the *Rate* Parameter

With SX/B version 1.5 the programming of interrupts is simplified with the *Rate* parameter. When used, the Rate parameter designates the interrupt frequency in calls per second. For example, to automate the timer code shown above at a rate of 200 calls per second, the INTERRUPT code could be modified as follows:

```
' -----------------------------------------------------------------------
  INTERRUPT 200
' -----------------------------------------------------------------------
ISR_Start:
  INC tix                     ' update tix counter
  IF tix = 200 THEN                ' check for 1 second
    tix = 0
    INC secs
    IF secs = 60 THEN
      secs = 0
      INC mins
      IF mins = 60 THEN
        mins = 0
        INC hrs
        IF hrs = MaxHr THEN
          hrs = 0
        ENDIF
      ENDIF
    ENDIF
  ENDIF

ISR_Exit:
  RETURNINT
```

By using this style the programmer does not need to set the OPTION register manually.

Asynchronous Interrupts

Every I/O pin in port B can be set to cause an interrupt upon logic level transitions (rising edge or falling edge). By configuring interrupts on input pins, the SX chip can respond to signal changes in a quick and deterministic fashion. In addition, an interrupt of this sort will wake up the SX chip from a SLEEP state. This can be configured by writing to the Edge Selection register (WKED_B) and the Wake-Up Enable register (WKEN_B) and detected by monitoring the Pending register (WKPND_B) in the interrupt routine. The I/O pins have interrupts disabled and are set to detect falling edge transitions by default.

As with edge selection, the Pending register bits will never be cleared by the SX alone; the running program is responsible for doing so. This means if a desired edge is detected, the interrupt will occur and the flag indicating this will remain set until the program clears it. Additional transitions on that pin will not cause interrupts until the associated bit in the Pending register is cleared.

To configure the I/O pins for wake-up (interrupt) edge detection:

1. Set I/O pin edge detection with WKED_B as desired.
2. Use WKEN_B to enable the individual pins for wake-up interrupts. A high bit (1) disables interrupts and a low bit (0) enables interrupts.
3. Set I/O pin directions as necessary.
4. Clear the Pending register to enable new interrupts.

5. Special Notes

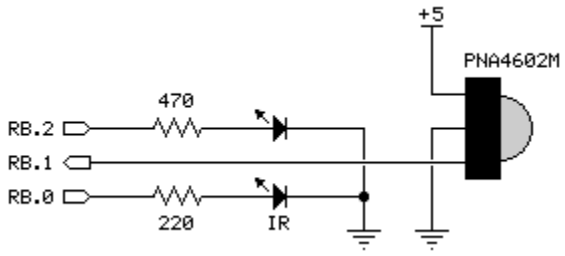
Interrupts that are not precisely timed should be avoided -- or at the very least disabled -- when using time-sensitive instructions such as **SERIN**, **SEROUT**, **PULSIN**, **PULSOUT**, etc. If an interrupt with variable timing is triggered while a time-sensitive instruction is active the instruction is likely to fail, or return bad results. If, however, the interrupt code is constructed such that is always runs a given number of cycles, the *EffectiveHz* parameter of **FREQ** may be used to allow time-sensitive SX/B instructions to operate while the interrupt is enabled.

Related projects: Clock / Timer and Quadrature Encoder Input

INTERRUPT Examples

Syntax

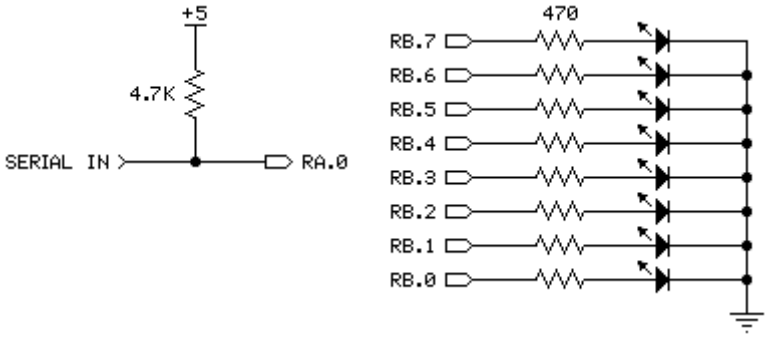
This example shows how to create an RTCC rollover interrupt to create the modulation frequency for an infrared LED.




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' This program modules an IR LED using the interrupt handler to toggle the
' IR LED state at a rate of 38.5 kHz. Since the program requires two
' changes of state at this rate, the ISR rate is doubled to 77,000.
'
' When an object reflects the IR from an the LED to the detector the Alarm
' LED will illuminate.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "ISR LED"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

IrLed             PIN   RB.0 OUTPUT       ' IR LED control
Detect            PIN   RB.1 INPUT        ' detector input
Alarm             PIN   RB.2 OUTPUT       ' alarm LED output

' -----------------------------------------------------------------------
 INTERRUPT NOPRESERVE 77_000
' -----------------------------------------------------------------------

ISR_Start:
  IrLed = ~IrLed                    ' toggle IR LED
  RETURNINT

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Program Code
```

```
' -----------------------------------------------------------------------

Start:
  LOW IrLed                                 ' make output, off
  LOW Alarm                                 ' make output, off

Main:
  Alarm = ~Detect                           ' check detector
  PAUSE 50
  GOTO Main
```

This example shows how to create an RTCC rollover interrupt to create a "background" serial receiver that runs up to 19.2 kBaud (with a 4 MHz clock), N81, true mode. A blinking LED indicates "foreground" activity while serial data is being received.

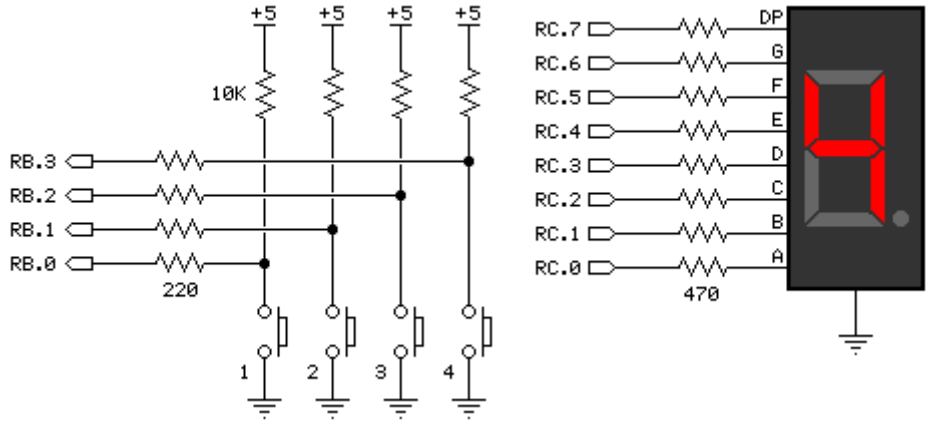



```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' This program demonstrates the construction of an ISR to receive serial
' data "in the background" using a 16-byte circular buffer.
'
' Note: Requires SX/B 1.2 or later

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4 000 000
ID          "ISR UART"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

Sin         PIN   RA.0 INPUT        ' serial in
Blinker     PIN   RB.0 OUTPUT       ' blinking LED
LEDs        PIN   RC   OUTPUT ' eight LEDs (7-segs)
TRIS_LEDs   VAR   TRIS_C

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------
```

```
B1200             CON   64                ' 1200 Baud
B2400             CON   32                ' 2400 Baud
B4800             CON   16                ' 4800 Baud
B9600             CON   8                 ' 9600 Baud
B19K2             CON   4                 ' 19.2 kBaud (max @ 4 MHz)

BitTm             CON   B19K2                   ' samples per bit
BitTm15     CON   3*BitTm/2         ' 1.5 bits

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

rxHead            VAR   Byte              ' available slot
rxTail            VAR   Byte              ' next byte to read
rxByte            VAR   Byte              ' serial byte
rxCount     vAR   Byte              ' bits to receive
rxTimer     VAR   Byte              ' bit timer for ISR
rxBuf             VAR   Byte(16)          ' circular buffer
tmpB1             VAR   Byte              ' parameter

' -----------------------------------------------------------------------------
 INTERRUPT NOPRESERVE
' -----------------------------------------------------------------------------

' ISR is setup to receive N81, true mode.

ISR_Start:
  ASM
  MOVB C, Sin                           ' sample serial input
  TEST rxCount                          ' receiving now?
  JNZ RX_Bit                            ' yes if rxCount > 0
  MOV W, #9                             ' start + 8 bits
  SC                                ' skip if no start bit
  MOV rxCount, W                    ' got start, load bit count
  MOV rxTimer, #BitTm15                 ' delay 1.5 bits

RX_Bit:
  DJNZ rxTimer, ISR_Exit                ' update bit timer
  MOV rxTimer, #BitTm                   ' reload bit timer
  DEC rxCount                           ' mark bit done
  SZ                                ' if last bit, we're done
  RR rxByte                             ' move bit into rxByte
  SZ                                ' if not 0, get more bits
  JMP ISR_Exit

RX_Buffer:
  MOV FSR, #rxBuf                       ' get buffer address
  ADD FSR, rxHead                       ' point to head
  MOV IND, rxByte                       ' move rxByte to head
  INC rxHead                            ' update head
  CLRB rxHead.4                     ' keep 0 - 15
  ENDASM

ISR_Exit:
  RETURNINT 52                          ' 13 uS @ 4 MHz


' =============================================================================
```

```
 PROGRAM Start
' ============================================================================

' ----------------------------------------------------------------------------
' Subroutine Declarations
' ----------------------------------------------------------------------------

GET_BYTE    FUNC  1           ' returns byte from buffer

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:
  TRIS_LEDs = %00000000                  ' make LED pins outputs
  OPTION = $88                           ' interrupt, no prescaler

Main:
  IF rxHead <> rxTail THEN               ' if buffer has data
    LEDs = GET_BYTE                      ' get byte from buffer
  ENDIF
  TOGGLE Blinker                    ' blink LED
  PAUSE 50                          ' small pause
  GOTO Main

' ----------------------------------------------------------------------------
' Subroutine Code
' ----------------------------------------------------------------------------

' Use: aVar = GET_BYTE
' -- if data is in buffer, the next byte is move to 'aVar'
' -- if called when buffer empty, code waits for character to arrive

FUNC GET_BYTE
  DO WHILE rxHead = rxTail
    ' wait while empty
  LOOP
  tmpB1 = rxBuf(rxTail)                  ' get first available
  INC rxTail                             ' update tail position
  \ CLRB rxTail.4                        ' keep 0 - 15
  RETURN tmpB1
ENDFUNC
```

This example shows how to use a port B edge detection interrupt to determine which button was pressed first.




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' On reset the 7-segment LED is cleared to a dash and the program waits
' for a button press (RB.0 - RB.3) -- the "winner" (first pressed) will
' be displayed on the display until reset.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "ISRPORTB"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

Buttons     PIN   RB INPUT          ' button inputs
Display     VAR   RC OUTPUT         ' 7-segment LED


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Dash        CON   %01000000         ' -
Dig1        CON   %00000110         ' 1
Dig2        CON   %01011011         ' 2
Dig3        CON   %01001111         ' 3
Dig4        CON   %01100110         ' 4
LtrE        CON   %01111001         ' E(rror)

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

winner            VAR   Byte              ' button pressed first
```

```
' -----------------------------------------------------------------------------
 INTERRUPT
' -----------------------------------------------------------------------------

ISR_Start:
  WKPND_B = winner                        ' get winner

Ch1: ' check channel
   IF winner <> %0001 THEN Ch2                ' if not, try next
  Display = Dig1                     ' otherwise display
  GOTO ISR_Exit

Ch2:
  IF winner <> %0010 THEN Ch3
  Display = Dig2
  GOTO ISR_Exit

Ch3:
  IF winner <> %0100 THEN Ch4
  Display = Dig3
  GOTO ISR_Exit

Ch4:
  IF winner <> %1000 THEN Uh_Oh
  Display = Dig4
  GOTO ISR_Exit

Uh_Oh:
  Display = LtrE                     ' something went wrong

ISR_Exit:
  WKEN_B = %11111111                      ' no ISR until reset
  RETURNINT

' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  Display = Dash                     ' indicate ready with dash
  WKPND_B = %00000000                     ' clear pending
  WKED_B = %11111111                      ' falling edge detect
  WKEN_B = %11110000                      ' use bits 0..3 for inputs
  END
```

LET

{**LET**} *Variable = {Value} {Op} Value*

Function
Assign a *Value* or result of an expression to *Variable*.

- ***Variable*** is the target variable for the assignment.
- ***Op*** is a unary (one value) or binary (two values) operator.
- ***Value*** is a variable or constant value which affects *Variable*

Explanation
LET is an optional keyword an not typcially used. For example,

```
LET idx = 25
```

... produces the same compiled output as:

```
idx = 25
```

so using **LET** is of no advantage. Note that when assigning a bit variable the value of a byte variable, the bit variable will be set to zero if the byte variable is zero, otherwise it will be set to one.

```
myByte = 0
  myBit = myByte                    ' myBit = 0
  myByte = 4
  myBit = myByte                    ' myBit = 1
```

Using Operators in Assignments
Note that SX/B supports only simple expressions, that is, just one operator per line of code. The following line will produce an error:

```
idx = count / 2 + 1 ' illegal in SX/B!
```

The error is corrected by splitting the operators across separate lines:

```
idx = count / 2 ' okay now  idx = idx + 1
```

The tables below show available operators for SX/B assignments:

Unary Operator	Definition
-	Negate
~	Bitwise inversion
NOT	Bitwise inversion

Binary Operator	Definition
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Modulus
*/	Multiply, return middle 16 bits
**	Multiply, return upper 16 bits
MAX	Set Maximum
MIN	Set Minimum
&	Bitwise AND
AND	Bitwise AND
|	Bitwise OR
OR	Bitwise OR
^	Bitwise Exclusive OR
XOR	Bitwise Exclusive OR
<<	Shift Left
SHL	Shift Left
>>	Shift Right
SHR	Shift Right

See the Operaters section for details.

Configuration Registers

When assigning values to SX configuration registers, the format is limited to:

{**LET**} *Register = Value*

When assigning values to the special registers WKPND_B and CMP_B a variable must be used and that variable will be exchanged with the SX register.

{**LET**} *Register = ByteVar*

```
WKPND_B = wakeUp              ' WKPND_B exchanged with wakeUp
  CMP_B = analog              ' CMP_B exchanged with analog
```

LOOKDOWN

LOOKDOWN *Target, Value0, {Value1, Value2, ...} Variable*

Function

Compare *Target* value to a list of values and store the index number of the first value that matches into *ByteVar*. If no value in the list matches, *Variable* is left unaffected.

- ***Target*** is a byte variable or constant (0 - 255) to be compared to the values in the list.
- ***Values*** are byte variables or constants (0 - 255) to be compared to *Target*.
- ***Variable*** is a byte or word variable that will be set to the index (0 - n) of the matching value in the *Values* list. If no matching value is found, *Variable* is left unaffected.

Explanation

LOOKDOWN works like the index in a book. In an index, you search for a topic and get the page number. **LOOKDOWN** searches for a target value in a list, and stores the index number of the first match in a variable. For example:

```
Do_Cmd:
  DO
    SERIN Sio, Baud, cmd                  ' wait for command input
    LOOKDOWN cmd, "F", "B", "S", cmd           ' compare against valid commands
  LOOP UNTIL cmd < 3                      ' wait until valid
  BRANCH cmd, Forward, Backward, Stop_Bot       ' execute valid command
```

Related instruction: LOOKUP

LOOKUP

Example

LOOKUP *Index, Value0, {Value1, Value2, ...} Variable*

Function

Find the value at location *Index* and store it in *ByteVar*. If *Index* exceeds the highest index value of the items in the list *ByteVar* is left unaffected.

- ***Index*** is a byte variable indicating the list item to retrieve. The first *Value* is at *Index* 0.
- ***Values*** are variables or constants.
- ***Variable*** is a byte or word variable that will be set to the value at the *Index* location.

Explanation

LOOKUP retrieves an item from a list based on the item's position, *Index*, in the list. For example:

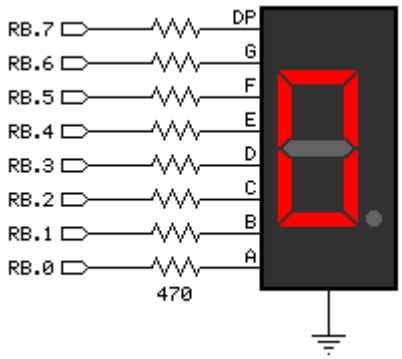
```
idx = 3
LOOKUP idx, $00, $01, $02, $04, $08, $10, $20, $40, $80, LEDs
```

In this example, the variable called LEDs will be set to $04 as this value appears at index position three in the values list.

Related instructions: LOOKDOWN and READ

LOOKUP Example

Syntax




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Uses LOOKUP to transfer a single digit value to a 7-segment LED.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          LOOKUP"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

LEDs        PIN   RB OUTPUT         ' 7-segment display


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

value             VAR   Byte              ' digit to display
tmpB1             VAR   Byte              ' subroutine parameter

' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------------

PUT_DIGIT   FUNC  1,1

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------
```

```
Start:


Main:
  DO
    FOR value = 0 TO 16                   ' demo values (last invalid)
      LEDs = PUT_DIGIT value              ' update the display
      PAUSE 1000                    ' pause one second
    NEXT
  LOOP

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' Use: destination = PUT_DIGIT value
' -- converts number in 'value' to 7-segment digit pattern
' and places it in 'destination'

FUNC PUT_DIGIT
  tmpB1 = __PARAM1                        ' copy value
  IF tmpB1 < $A THEN
    ' decimal
    LOOKUP tmpB1, $3F,$06,$5B,$4F,$66,$6D,$7D,$07,$7F,$67, tmpB1
  ELSE
    IF tmpB1 < $10 THEN
      ' hex
      tmpB1 = tmpB1 - 10                  ' adjust for LOOKUP
      LOOKUP tmpB1, $77,$7C,$39,$5E,$79,$71, tmpB1
    ELSE
      tmpB1 = %01000000                   ' display dash
    ENDIF
  ENDIF
  RETURN tmpB1
ENDFUNC
```

LOW

Example

LOW *Pin*

Function
Make the specified *Pin* an output and low (0).

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).

Explanation
The **LOW** instruction makes the specified *Pin* an output, and then sets its value to 0 (Vss). For example:

```
LOW RA.3
```

Does the same thing as:

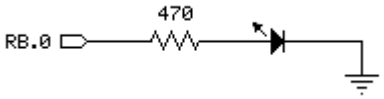
```
OUTPUT RA.3
RA.3 = 0
```

Using the **LOW** instruction is more convenient in this case.

LOW can be used on a whole port; for example **LOW RB** will make all pins on port RB low.

Related instructions: HIGH, OUTPUT, and TOGGLE

HIGH / LOW Example




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Simple LED blinker using HIGH and LOW.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "HIGH-LOW"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

LED         PIN   RB.0  OUTPUT      ' LED pin

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

OnDelay     CON   150         ' time LED is on
OffDelay    CON   350         ' time LED is off

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  DO
    HIGH LED                        ' turn LED on
    PAUSE OnDelay                   ' delay
    LOW LED                         ' turn LED off
    PAUSE OffDelay                  ' delay
  LOOP                              ' repeat forever
```

ON ...

ON *Index* [**GOTO** | **GOSUB**] *Label0 {, Label1, Label2, ...}*
ON *Index = Value0 {, Value1, Value2, ...}* [**GOTO** | **GOSUB**] *Label0 {, Label1, Label2, ...}*

Function
Jump to the address specified by *Index* (if in range).

- ***Index*** is a byte variable that specifies the index of the address, in the list, to jump to (0 - N), or is value to be compared against a list to create the jump offset.
- ***Label*** specifies the location to jump to.

Quick Facts

	All SX Models
Limit of *Label* entries	256

Explanation
The **ON...** instruction is useful when you want to write something like this:

```
IF idx = 0 THEN Case_0            ' idx = 0: jump to label "Case_0"
IF idx = 1 THEN Case_1            ' idx = 1: jump to label "Case_1"
IF idx = 2 THEN Case_2            ' idx = 2: jump to label "Case_2"
```

You can use **ON...** with **GOTO** to organize this into a single statement:

```
ON idx GOTO Case_0, Case_1, Case_2
```

This works exactly the same as the previous **IF...THEN** example. If the value isn't in range (in this case if value is greater than 2), **ON...GOTO** does nothing and the program continues with the next instruction after **ON...GOTO**. **ON...GOSUB** is identical, however, the program should be designed such that a **RETURN** is placed at the end of the code at *Label*.

A variant of the **ON...** instruction essentially combines **LOOKDOWN** into the syntax. For example:

```
Main:
  cmd = RX_BYTE
  LOOKDOWN cmd, "L", R", "S", cmd
  ON cmd GOSUB Robot_Left, Robot_Right, Robot_Stop
  GOTO Main
```

... can be combined to:

```
Main:
  cmd = RX_BYTE
  ON cmd = "L", R", "S" GOSUB Robot_Left, Robot_Right, Robot_Stop
  GOTO Main
```

Related instructions: IF ... THEN and BRANCH

OUTPUT

OUTPUT *Pin*

Function
Make the specified *Pin* an output by writing a zero (0) to the corresponding bit of the associated port tris register.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).

Explanation
There are several ways to make a pin an output. When an SX/B program is reset, all of the I/O pins are made inputs. Instructions that rely on output pins, like **PULSOUT** and **SEROUT**, automatically change the specified pin to output mode. Writing 0's to particular bits of the port TRIS register makes the corresponding pins outputs. And then there's the **OUTPUT** instruction:

```
OUTPUT RA.3
RA.3 = 0
```

When your program changes a pin from input to output, whatever state happens to be in the corresponding bit of port TRIS register sets the initial state of the pin. To simultaneously make a pin an output and set its state use the **HIGH** and **LOW** instructions.

OUTPUT can also be used on a whole port; for example **OUTPUT RB** will make all pins on port RB outputs.

Related instructions: INPUT and REVERSE

OWRDBIT

OWRDBIT *Pin*, *Bit Var*

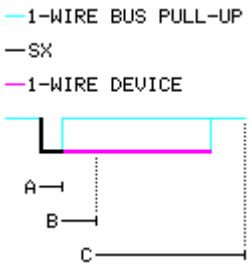
Function
Reads one bit from a 1-Wire bus.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***BitVar*** is a bit variable that will hold the 1-Wire Read Slot value.

Quick Facts

	All SX Models
Special Notes	The DQ pin (specified by *Pin*) must have a 4.7 kΩ pull-up resister.
	1-Wire commands require a FREQ setting of 4 MHz or higher.

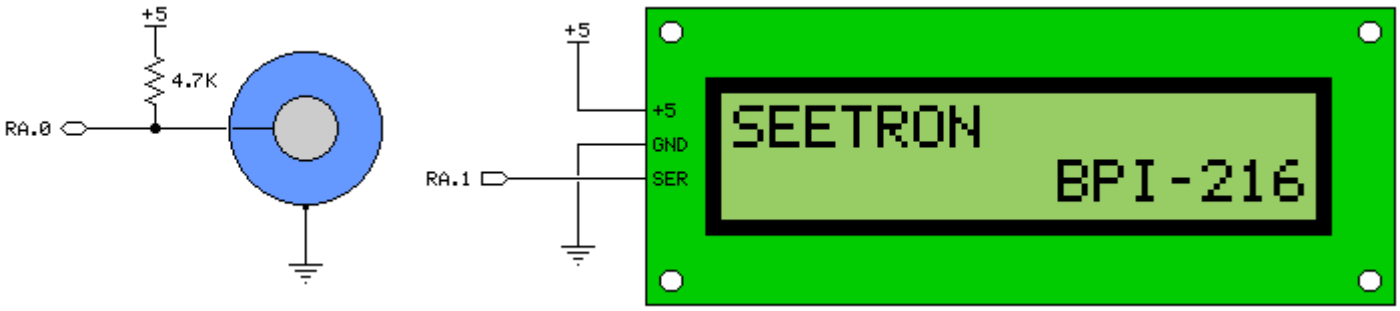
Explanation
Some 1-Wire transactions require reading data from the device. While this is typically handled with **OWRDBYTE**, SX/B supports bit access with the **OWRDBIT** instruction. A 1-Wire read is accomplished by generating a brief (5 µS) low-pulse and sampling the DQ pin within 15 µS of the falling edge of the pulse. This is called a "Read Slot." The diagram and table below details the Read Slot sequence generated by the SX with **OWRDBIT**.



Note	Timing	Description
A	5 µS	Initiate Read Slot (SX master)
B	10 µS	Delay before sampling DQ
C	~ 60 µS	Read Slot recovery

Related instructions: OWRESET, OWRDBYTE, OWWRBIT, and OWWRBYTE

1-Wire Example




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' This program scans the 1-Wire bus and when a device is detected it will
' read and display its serial number.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "1-WIRE"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

DQ          PIN   RA.0              ' 1-Wire bus (4.7k pull-up)
SOut        PIN   RA.1 OUTPUT       ' output to SEETRON 2x16

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Baud        CON   "N9600"

LcdI        CON   $FE               ' instruction
LcdCls            CON   $01               ' clear the LCD
LcdHome     CON   $02               ' move cursor home
LcdCrsrL    CON   $10               ' move cursor left
LcdCrsrR    CON   $14               ' move cursor right
LcdDispL    CON   $18               ' shift chars left
LcdDispR    CON   $1C               ' shift chars right
LcdDDRam    CON   $80               ' Display Data RAM control
LcdCGRam    CON   $40               ' Character Generator RAM
LcdLine1    CON   $80               ' DDRAM address of line 1
LcdLine2    CON   $C0               ' DDRAM address of line 2
```

```
SearchROM   CON   $F0
ReadROM     CON   $33              ' read ID, serial num, CRC

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

idx         VAR   Byte             ' loop counter
char        VAR   Byte             ' char for LCD
owByte      VAR   Byte             ' byte for OW work
owSerial    VAR   Byte(8)          ' OW serial number
tmpB1       VAR   Byte             ' subroutine work vars
tmpB2       VAR   Byte
tmpB3       VAR   Byte
tmpW1       VAR   Word

' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------------

DELAY       SUB   1, 2             ' delay in milliseconds
TX_BYTE     SUB   1, 2             ' transmit byte
TX_STR      SUB   2                ' transmit string
TX_HEX2     SUB   1                ' transmit byte as HEX2

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  PLP_A = %0011                    ' enable pull-ups
  PLP_B = %00000000                ' .. except RA.0 and RA.1
  PLP_C = %00000000
  DELAY 750                        ' let LCD initialize

Main:
  TX_BYTE LcdI                     ' clear screen, home cursor
  TX_BYTE LcdCls
  DELAY 1
  TX_STR "SX/B 1-Wire Demo"        ' write banner in LCD

Scan_DQ:
  TX_BYTE LcdI
  TX_BYTE LcdLine2
  OWRESET DQ, owByte               ' reset and sample presence
  ON owByte GOTO Bus_Short, Bad_Bus, Found, No_Device
  GOTO Main

Bus_Short:
  TX_STR "1-Wire bus short"
  DELAY 1000
  GOTO Scan_DQ

Bad_Bus:
  TX_STR "Bad connection "
  DELAY 1000
```

```
  GOTO Scan_DQ

Found:
  TX_STR "Found device "
  DELAY 500
  TX_BYTE LcdI                      ' move back to L2/C0
  TX_BYTE LcdLine2
  OWRESET DQ, owByte                ' reset device
  OWWRBYTE DQ, ReadROM              ' send read ROM command
  FOR idx = 0 TO 7                  ' read serial number
    OWRDBYTE DQ, owByte
    owSerial(idx) = owByte
  NEXT
  FOR idx = 7 TO 0 STEP -1          ' display on LCD
    owByte = owSerial(idx)
    TX_HEX2 owByte                  ' -- hex mode
  NEXT
  DELAY 2000
  GOTO Scan_DQ

No_Device:
  TX_STR "No device "
  GOTO Scan_DQ

' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                ' save byte value
  ELSE
    tmpW1 = __WPARAM12              ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB

' -----------------------------------------------------------------------------
' Use: TX_BYTE theByte {, count}
' -- transmit "theByte" at "Baud" on "SOut"
' -- optional "count" may be specified (must be > 0)

SUB TX_BYTE
  tmpB1 = __PARAM1                  ' save byte
  IF __PARAMCNT = 1 THEN            ' if no count
    tmpB2 = 1                       ' set to 1
  ELSE                              ' otherwise
    tmpB2 = __PARAM2                ' get count
    IF tmpB2 = 0 THEN               ' do not allow 0
      tmpB2 = 1
    ENDIF
  ENDIF
  DO WHILE tmpB2 > 0                ' loop through count
    SEROUT SOut, Baud, tmpB1        ' send the byte
    DEC tmpB2                       ' decrement count
  LOOP
ENDSUB
```

```
' -----------------------------------------------------------------------------
' Use: TX_STR [string | label]
' -- "string" is an embedded string constant
' -- "label" is DATA statement label for stored z-String

SUB TX_STR
  tmpW1 = __WPARAM12 ' get offset/base
  DO
    READ tmpW1, char ' read a character
    IF char = 0 THEN EXIT ' if 0, string complete
    TX_BYTE char ' send character
    INC tmpW1 ' point to next character
  LOOP
ENDSUB
' -----------------------------------------------------------------------------
' Use: TX_HEX2 theByte
' -- transmit 'theByte' in HEX2 format

SUB TX_HEX2
  tmpB3 = __PARAM1                  ' save byte
  char = tmpB3 & %11110000          ' mask high nib
  SWAP char                         ' swap nibs
  READ HexDigits + char, char       ' high nib to hex char
  TX_BYTE char                      ' send it
  char = tmpB3 & %00001111          ' get low nib
  READ HexDigits + char, char       ' low nib to hex char
  TX_BYTE char                      ' send it
ENDSUB

' =============================================================================
' User Data
' =============================================================================

HexDigits:
  DATA "0123456789ABCDEF"
```

OWRDBYTE

Example

OWRDBYTE *Pin*, *ByteVar*

Function
Reads one byte (eight bits) from a 1-Wire device.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***ByteVar*** is any byte variable that will hold the 1-Wire read value.

Quick Facts

	All SX Models
Special Notes	The DQ pin (specified by *Pin*) must have a 4.7 kΩ pull-up resister.
	1-Wire commands require a FREQ setting of 4 MHz or higher.
Receive Rate	~12 kBits/Sec (not including Reset)

Explanation
Some 1-Wire transactions require reading data from the device. A bit is read from the 1-Wire device byte generating a brief pulse on the DQ line, then reading the line within 15 µS of the falling edge (see **OWRDBIT** for details). This is called a "Read Slot." The **OWRDBYTE** instruction generates eight 1-Wire Read Slot sequences and returns the value in *ByteVar*.

Related instructions: OWRESET, OWRDBIT, OWWRBIT, and OWWRBYTE

OWRESET

Example

OWRESET *Pin* {, *ByteVar*}

Function

Generates a 1-Wire reset sequence on *Pin*, returning (optional) status information in *ByteVar*.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***ByteVar*** is an optional byte variable that will hold the status of the 1-Wire connections (see below).

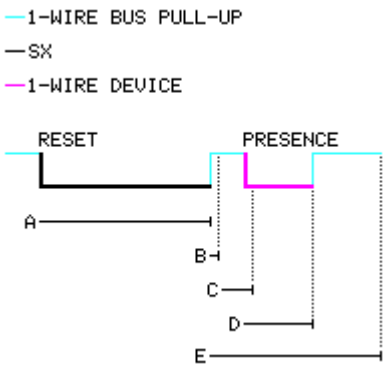
Quick Facts

	All SX Models
Special Notes	The DQ pin (specified by *Pin*) must have a 4.7 kΩ pull-up resister.
	1-Wire commands require a FREQ setting of 4 MHz or higher.
Status value	0 = Bus shorted to Vss; 1 = Bad connection; 2 = Good connection; 3 = No device

Explanation

All transactions on the 1-Wire bus begin with an Initialization sequence that consists of a Reset pulse generated by the master, followed by a Presence pulse generated by the 1-Wire slave. The **OWRESET** instruction generates the 1-Wire Reset pulse on the specified DQ *Pin* and, if *ByteVar* is specified, will monitor the bus and return status information to the program.

The diagram and table below details the Reset pulse generated by the SX with **OWRESET** and a typical Presence pulse generated by a 1-Wire slave, in response.



Note	Timing	Description
A	~ 480 µS	1-Wire Reset pulse (SX master)
B	~ 15 µS	Delay before sampling for bus short
C	~ 70 µS	Delay before sampling for Presence
D	~ 60 - 240 µS	Presence pulse (1-Wire slave)
E	~ 480 µS	Reset recovery

Related instructions: OWRDBIT, OWRDBYTE, OWWRBIT, and OWWRBYTE

OWWRBIT

OWWRBIT *Pin*, *BitVal*

Function
Writes one bit to a 1-Wire bus.

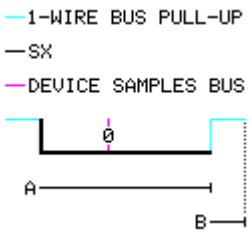
- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***BitVal*** is a variable or constant value (0 to 1) that will be written to the 1-Wire bus.

Quick Facts

	All SX Models
Special Notes	The DQ pin (specified by *Pin*) must have a 4.7 kΩ pull-up resister.
	1-Wire commands require a FREQ setting of 4 MHz or higher.

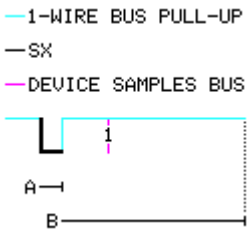
Explanation
After reset, 1-Wire transactions require writing values to the bus. While this is typically handled with **OWWRBYTE**, SX/B supports bit access with the **OWWRBIT** instruction. A bit is written by generating a timed low pulse on the DQ line; this is called a "Write Slot." The diagrams and tables below details the Write Slot sequences generated by the SX with **OWWRBIT**.

WRITE 0



Note	Timing	Description
A	60 µS	Write Slot 0 (SX master)
B	~ 10 µS	Write Slot recovery

WRITE 1



Note	Timing	Description
A	5 µS	Write Slot 1 (SX master)
B	~ 65 µS	Write Slot recovery

Related instructions: OWRESET, OWRDBIT, OWRDBYTE, and OWWRBYTE

OWWRBYTE

Example

OWWRBYTE *Pin*, *ByteVal*

Function
Writes one byte (eight bits) to the 1-Wire bus.

- ***Pin*** is any SX IO pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***ByteVal*** is a variable or constant value (0 - 255) that will be written to the 1-Wire bus.

Quick Facts

	All SX Models
Special Notes	The DQ pin (specified by *Pin*) must have a 4.7 kΩ pull-up resister.
	1-Wire commands require a FREQ setting of 4 MHz or higher.
Transmission rate	~12 kBits/Sec (not including Reset)

Explanation
After reset, 1-Wire transactions require writing values to the bus. A bit is written by generating a timed low pulse on the DQ line; this is called a "Write Slot" (see **OWWRBIT** for details). The **OWWRBYTE** instruction generates eight Write Slot sequences to put *ByteVal* on the 1-Wire bus.

Related instructions: OWRESET, OWRDBIT, OWRDBYTE, and OWWRBIT

PAUSE

Example

PAUSE *Value*
PAUSE *ByteVal1 {[, | *] ByteVal2}*

Function
Pause the program (do nothing) for a number of milliseconds.

- ***Value*** is a byte or word variable/constant (1 to 65535), or fractional constant value†.
- ***ByteVal*** is a variable (1 - 255) or constant (1 to 255).

† **Note:** Fractional values (0.01 to 65535.99) are allowed when a single constant parameter is used. SX clock speed will affect the accuracy of fractional timing.

Explanation
PAUSE delays the execution of the next program instruction for a number of milliseconds based on *Value*, or *ByteVal1* and *ByteVal2*. For example:

```
Flash:
  DO
    LOW RC.0
    PAUSE 1000
    HIGH RC.0
    PAUSE 1000
  LOOP
```

This code causes pin RC.0 to go low for 1000 milliseconds, then high for 1000 milliseconds. Note that a **PAUSE** duration of up to 65535 milliseconds is possible with a 16-bit variable or constant.

Some projects may require sub- or fractional-millisecond delays, and in some cases **PAUSEUS** may not be practical or desired. When using a single parameter, a fractional **PAUSE** value may be specified:

```
Flash:
  DO
    LOW RC.0
    PAUSE 12.5
    HIGH RC.0
    PAUSE 12.5
  LOOP
```

PAUSE can take a single byte or word parameter, or two byte parameters. When using two byte parameters, two forms are allowed. Here's an example of the first form:

```
PAUSE 238, 2
```

Using this form, the **PAUSE** duration is *ByteVal1* + (*ByteVal2** 256). In the example above the program will pause for 750 milliseconds. Note that this form is typically used with variables, but will also work with constants up to 255. Note that as of SX/B version 1.5 using a single word variable or constant is easier to code and understand.

The third form of **PAUSE** is demonstrated below:

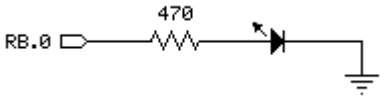
```
idx = 3   PAUSE idx * 250
```

Using this form, the **PAUSE** duration is *ByteVal1* * *ByteVal2*. In the example above the program will pause for 750 milliseconds. Note that this form is typically used with variables, but will also work with constants up to 255.

Related instruction: PAUSEUS

PAUSE Example

Syntax




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Variable-rate LED blinker. Puts PAUSE into a subroutine for greatest
' flexibility and program space conservation.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "PAUSE"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

Led         PIN   RB.0 OUTPUT

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

idx         VAR   Byte              ' loop counter
timing            VAR   Word
tmpW1             VAR   Word              ' for subroutines

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

DELAY             SUB   1, 2             ' delay in milliseconds

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  DO
    FOR idx = 1 TO 10                   ' loop 1 to 10
      HIGH Led                          ' LED on
      timing = 50 * idx                 ' calculate on timing
      DELAY timing                      ' pause 50 to 500 ms
```

```
      LOW Led                           ' LED off
      timing = 50 * idx
      DELAY timing
    NEXT
  LOOP

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                    ' save byte value
  ELSE
    tmpW1 = __WPARAM12                  ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB
```

PAUSEUS

PAUSEUS *Value*
PAUSEUS *ByteVal1 {[, | *] ByteVal2}*

Function
Pause the program (do nothing) for a number of microseconds.

- ***Value*** is a byte or word variable/constant (1 to 65535), or fractional constant value†.
- ***ByteVal*** is a variable (1 - 255) or constant (1 to 255).

† **Note:** Fractional values (0.01 to 65535.99) are allowed when a single constant parameter is used. SX clock speed will affect the accuracy of fractional timing.

Explanation
PAUSE delays the execution of the next program instruction for a number of microseconds based on *Value*, or *ByteVal1* and *ByteVal2*. For example:

```
Tone:
  RC.0 = ~RC.0
  PAUSEUS 500
  IF RB.0 = 1 THEN Tone
```

This code toggles the state of RC.0 every 500 microseconds, creating a 1 kHz tone until pin RB.0 goes low. Note that a **PAUSEUS** duration of up to 65535 microseconds is possible when a single constant parameter is used.

Some projects may require sub- or fractional-microsecond delays. When using a single parameter, a fractional **PAUSEUS** value may be specified:

```
Tone440:
  DO
    RC.0 = ~RC.0
    PAUSEUS 1136.36
  LOOP
```

PAUSEUS can take a single byte or word parameter, or two byte parameters. When using two byte parameters, two forms are allowed. Here's an example of the first form:

```
PAUSEUS 238, 2
```

Using this form, the **PAUSEUS** duration is *ByteVal1* + (*ByteVal2** 256). In the example above the program will pause for 750 microseconds. Note that this form is typically used with variables, but will also work with constants up to 255. Note that as of SX/B version 1.5 using a single word variable or constant is easier to code and understand.

The third form of **PAUSEUS** is demonstrated below:

```
idx = 3
PAUSEUS idx * 250
```

Using this form, the **PAUSEUS** duration is *ByteVal1* * *ByteVal2*. In the example above the program will pause for 750 microseconds. Note that this form is typically used with variables, but will also work with constants up to 255.

Related instruction: PAUSE

PULLUP (SX48/52 Only)

PULLUP *Pin {, Enable}*

Function

Enables the internal pull-up resistor for *Pin* on the SX48 or SX52. This command does not apply to the SX18, SX20, or SX28 (use the PLP_A, PLP_B, and PLP_C registers).

- ***Pin*** is any SX48/52 I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***Enable*** is a constant, 0 or 1, that enables (1) or disables (0) the internal pull-up resistor for *Pin*. When not specified, *Enable* defaults to 1.

Explanation

Every I/O pin has an optional internal pull-up resister that can be configured by writing to the appropriate pull-up register (PLP_A, PLP_B, PLP_C, PLP_D and PLP_E). By configuring pull-up resisters on input pins, the SX chip can be connected directly to open/drain circuitry without the need for external pull-up resisters. The internal pull-up resisters are disabled by default. Pull-up resisters can be activated for all pins, regardless of pin direction but really matter only when the associated pin is set to input mode.

```
Start:
  PULLUP RA.0, 1              ' enable pull-up
  PULLUP RA.1                      ' enable pull-up
  PULLUP RA.2, 0              ' disable pull-up
```

Related instructions: CMOS, TTL, and SCHMITT

PULSIN

PULSIN *Pin, State, Variable {, Resolution}*

Function

Measure the width of a pulse on *Pin* described by *State* in units of *Resolution* and store the result in *Variable*.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7). This pin will be set to input mode.
- ***State*** is a constant (0 - 1) that specifies whether the pulse to be measured is low (0) or high (1). A low pulse begins with a 1-to-0 transition, and a high pulse begins with a 0-to-1 transition.
- ***Variable*** is a byte or word variable in which the measured pulse duration will be stored. The unit of time for *Variable* is described in *Resolution*.
- ***Resolution*** is an optional constant (1 - 255) that specifies the units for *Variable*, in increments of 10 microseconds (default value is 1 when not specified).

Quick Facts

Range of *Variable*	0 (no pulse) to 255 (byte) or 65,535 (word)
Units in *Resolution*	10 µs (0.01 ms)
Minimum pulse width	10 µs
Maximum pulse width	650.25 ms (byte), 167.11 s (word)

Explanation

PULSIN is like a fast stopwatch that is triggered by a change in state (0 or 1) on the specified pin. The entire width of the specified pulse (high or low) is measured, in units shown above and stored in ByteVar.

Many analog properties (voltage, resistance, capacitance, frequency, duty cycle) can be measured in terms of pulse duration. This makes **PULSIN** a valuable form of analog-to-digital conversion.

PULSIN will wait for the desired pulse, for up to the maximum pulse width it can measure, shown in the table above. If it sees the desired pulse it measures the time until the end of the pulse and stores the result in *Variable*. If it never sees the start of the pulse, or the pulse is too long (greater than the Maximum Pulse Width shown above), **PULSIN** "times out" and store 0 in *Variable*. This operation keeps your program from locking-up should the desired pulse never occur.

```
PULSIN RA.0, 1, pWidth ' measure 10 - 2550 us pulse
```

In the example above, pin RA.0 will be set to input mode, wait for a low-to-high transition, then measure the period that the pin stays high. Using the default Resolution of 10 microseconds, a pulse from 10 microseconds to 2.55 milliseconds in width can be measured (assuming *pWidth* is a byte).

Since **PULSIN** uses a byte variable for storage, the *Resolution* parameter can be used to allow the measurement of wider pulse widths. For the best accuracy, set *Resolution* to the smallest value that will allow the measurement of the greatest expected pulse width.

```
SUB Get_Button
  PULSIN RA.1, 0, btnIn, 100        ' wait for (low) button press
  IF btnIn < 50 THEN Get_Button     ' at least 50 ms?
ENDSUB
```

The subroutine above will monitor the state of RA.1, waiting for it to go low and stay low for at least 50 milliseconds. This is a useful method of debouncing a button input.

Related instruction: PULSOUT
Related project: SIRCS Decoder

PULSOUT

PULSOUT *Pin, Duration {, Resolution}*

Function
Generate a pulse on *Pin* with a width of *Duration* * *Resolution*.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7). This pin will be set to output mode.
- ***Duration*** is a variable or constant that specifies the pulse width in *Resolution* units.
- ***Resolution*** is an optional constant (1 - 255) that specifies the units for *Duration*, in increments of 10 microseconds (default value is 1 when not specified).

Quick Facts

Units in *Resolution*	10 µs (0.01 ms)
Minimum pulse width	10 µs
Maximum pulse width	650.25 ms (byte), 167.11 s (word)

Explanation
PULSOUT sets *Pin* to output mode, inverts the state of that pin; waits for the specified *Duration x Resolution x 10* microseconds; then inverts the state of the pin again returning the bit to its original state.

The pulse width is the product of *Duration* and *Resolution*. In the following example, a pulse of 50 microseconds will be generated on RA.0

```
PULSOUT RA.0, 5 ' 50 µs pulse
```

The *Resolution* parameter gives the programmer a great deal of flexibility with **PULSOUT**. In the follow example, a stream of pulses will be created, each with a different pulse width:

```
Main:
  FOR idx = 1 TO 10
    PULSOUT RA.0, idx, 10                ' pulses from 100 to 1000 µs
    PAUSE 1                              ' delay 1 millisecond
  NEXT
  GOTO Main
```

By combining *Duration* and *Resolution*, pulses from 10 microseconds to 167.11 seconds can be generated.

Related instruction: PULSIN

PUT

PUT *Location, Value {, Value, ... }*

Function
Copy *Value(s)* into RAM, beginning at *Location*.

- ***Location*** is the starting address in RAM for the **PUT**.
- ***Value*** is a variable, constant value, or string constant.

Explanation
The **PUT** instruction provides a convenient method of moving multiple values into consecutive locations in RAM. For example,

```
clock(0) = hrs                      ' copy hrs to clock(0)
clock(1) = mins                  ' copy mins to clock(1)
clock(2) = secs                  ' copy mins to clock(2)
```

Can be simplified to a single line of code:

```
PUT clock(0), hrs, mins, secs
```

Note that array elements are internally represented by RAM addresses. To use **PUT** with consecutive bytes that are not part of an array, you must explicitly declare the address of the first variable using the @ symbol:

```
PUT @hrs, 12, 34, 56 ' hrs = 12, mins = 34, secs = 56
```

When *Value* is a word, its LSB is written from *Location* and its MSB to *Location* + 1. For example:

```
loVal VAR Byte
hiVal VAR Byte

Start:
  PUT @loVal, $F00A                 ' loVal = $0A, hiVal = $F0
```

PUT can be used to load string characters into a byte array:

```
 PUT msg, "Hello", 0
```

Using PUT With Address Parameters

When a variable is passed to a subroutine (declared with SUB) by address (using @), as in:

```
Some Routine @aValue ' pass address of 'aValue'
```

... **PUT** can be used to place a value into the address that was passed to the subroutine:

```
Some_Routine:
  rtnAddr = __PARAM1                ' save return address
  GET rtnAddr, theValue             ' get value from address
  ...                               ' do something with the value
  PUT rtnAddr, newValue             ' update value at passed address
  RETURN
```

This technique is useful for allowing a subroutine to modify a variable. Note that for single-parameter instances, the __RAM() system array may be used in place of **GET** and **PUT:**

```
Some_Routine:
  rtnAddr = __PARAM1                ' save return address
  theValue = __RAM(rtnAddr)         ' get value from address
  ...                               ' do something with the value
  __RAM(rtnAddr) = newValue         ' update value at passed address
  RETURN
```

Related instruction: GET

PWM

Example

PWM *Pin, Duty, Duration*

Function

Convert a digital value to analog output via pulse-width modulation.

- ***Pin*** is any SX IO pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***Duty*** is a variable or constant (0 - 255) that specifies the analog output level (0 to 5V).
- ***Duration*** is a variable or constant (0 - 255) that specifies the duration of the PWM output in milliseconds.

Quick Facts

Units in *Duration*	1 ms
Average voltage equation	Average Voltage = (*Duty* ÷ 255) x 5 volts
Duration	5 x R x C (suggested)
Notes	*Pin* is output while PWM active, then switched to input mode

Explanation

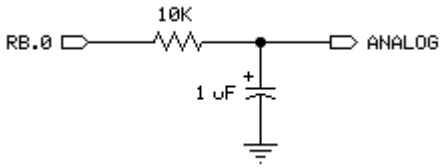
Pulse-width modulation (**PWM**) allows the SX (a purely digital device) to generate an analog voltage. The basic idea is this: If you make a pin output high, the voltage at that pin will be close to 5V. Output low is close to 0V. What if you switched the pin rapidly between high and low so that it was high half the time and low half the time? The average voltage over time would be halfway between 0 and 5V (2.5V). **PWM** emits a burst of 1s and 0s with a ratio that is proportional to the duty value you specify.

The proportion of 1s to 0s in **PWM** is called the duty cycle. The duty cycle controls the analog voltage in a very direct way; the higher the duty cycle the higher the voltage. In the case of the SX, the duty cycle can range from 0 to 255. *Duty* is literally the proportion of 1s to 0s output by the **PWM** instruction. To determine the proportional **PWM** output voltage, use this formula: (*Duty* ÷ 255) x 5V. For example, if *Duty* is 100, (100 ÷ 255) x 5V = 1.96V; **PWM** outputs a train of pulses to create (through an RC network) an average voltage of 1.96V.

In order to convert **PWM** into an analog voltage we have to filter out the pulses and store the average voltage. The resistor/capacitor combination shown below will do the job. The capacitor will hold the voltage set by **PWM** even after the instruction has finished. How long it will hold the voltage depends on how much current is drawn from it by external circuitry, and the internal leakage of the capacitor. In order to hold the voltage relatively steady, a program must periodically repeat the **PWM** instruction to give the capacitor a fresh charge.

Just as it takes time to discharge a capacitor, it also takes time to charge it in the first place. The **PWM** command lets you specify the charging time in terms of **PWM** duration. The timing for the units in *Duration* is one millisecond.

How do you determine how long to charge a capacitor? Use this rule-of-thumb formula: Charge Time = 5 x R x C. For instance, the circuit below uses a 10 kΩ (10 x 10^3 ohm) resistor and a 1 µF (1 x 10^{-6} uF) capacitor:



Charge time = 5 x 10 x 10^3 x 1 x 10^{-6} = 50 x 10^{-3} seconds, or 50 milliseconds.

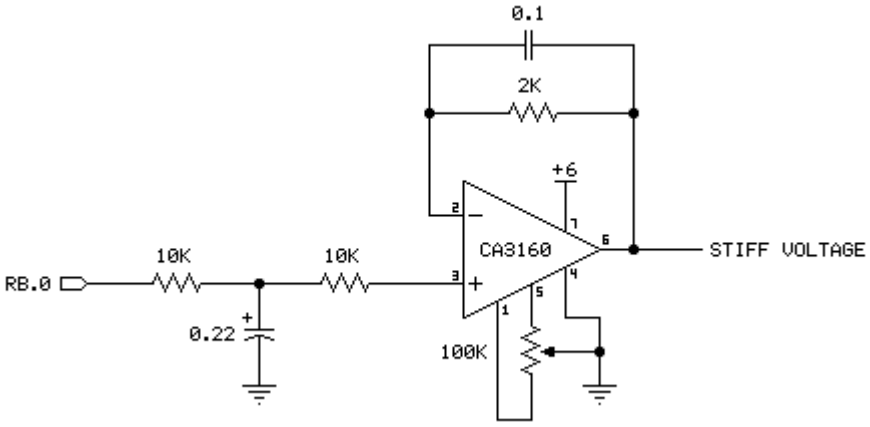
```
PWM RB.0, 128, 50 ' charge to 2.5 volts, 50 ms
```

After outputting the **PWM** pulses, the SX leaves the pin in input mode (1 in the corresponding bit of the pin's TRIS register). In input mode, the pin's output driver is effectively disconnected. If it were not, the steady output state of the pin would change the voltage on the capacitor and undo the voltage setting established by **PWM**. Keep in mind that leakage currents of up to 1 µA can flow into or out of this "disconnected" pin. Over time, these small currents will cause the voltage on the capacitor to drift. The same applies for leakage current from an op-amp's input, as well as the capacitor's own internal leakage. Executing **PWM** occasionally will reset the capacitor voltage to the intended value.

PWM charges the capacitor; the load presented by your circuit discharges it. How long the charge lasts (and therefore how often your program should repeat the **PWM** instruction to refresh the charge) depends on how much current the circuit draws, and how stable the voltage must be. You may need to buffer **PWM** RC circuit output with a simple op-amp follower if your load or stability requirements are more than the passive circuit can handle.

PWM Example

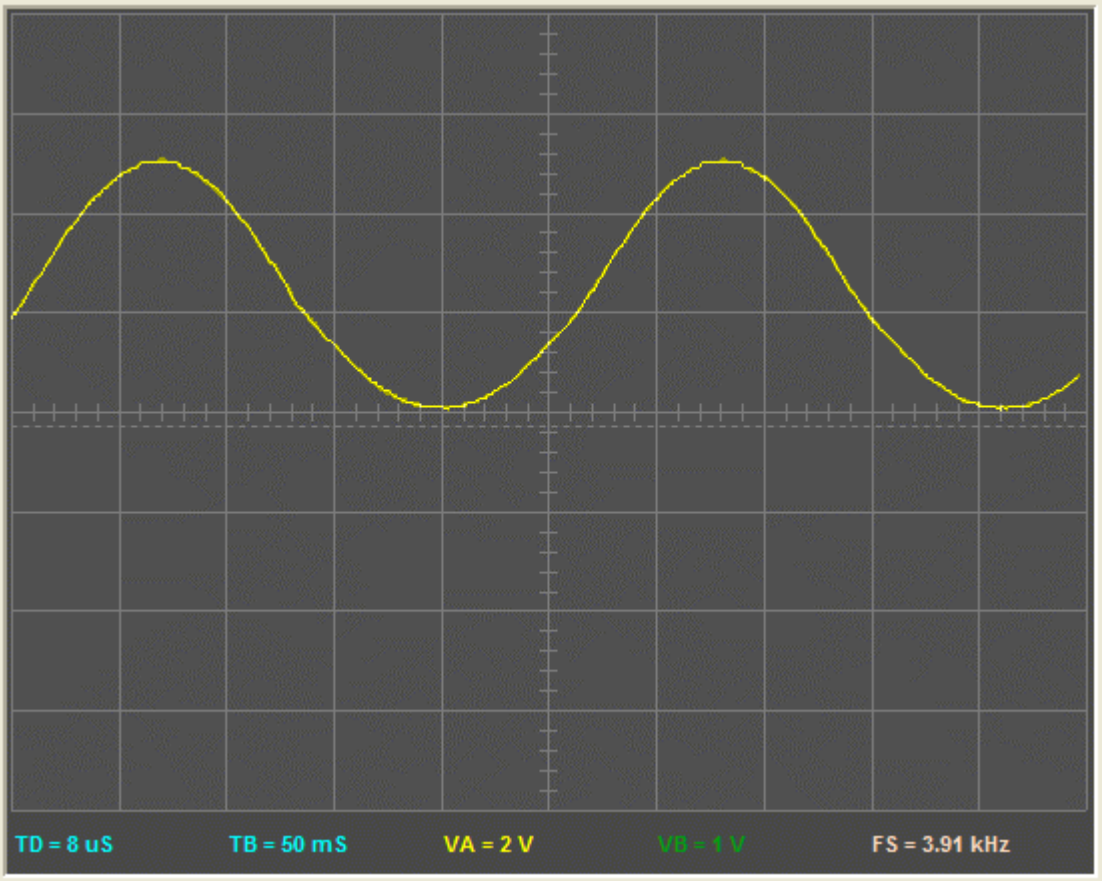
Syntax




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Uses PWM through an RC network to create an analog voltage.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID    PWM"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

DAC         PIN   RB.0 OUTPUT       ' output to RC network

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

angle             VAR   Byte
duty        VAR   Byte
tmpB1             VAR   Byte
tmpB2             VAR   Byte

' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------------

SIN         FUNC  1,1               ' returns sine of angle
COS         FUNC  1,1               ' returns cosine of angle
```

```
' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  DO
    FOR angle = 0 TO 255            ' sweep through angles
      duty = SIN angle              ' get sine of angle
      duty = duty + 127             ' bias to 2.5 v
      PWM DAC, duty, 1              ' charge RC network
    NEXT
  LOOP

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' Use: value = SIN angle
' -- "value" returned as signed byte (-127 to 127)
' -- angle is expressed in Brads (0 - 255)

FUNC SIN
  tmpB1 = __PARAM1                  ' get angle
  tmpB2 = tmpB1                     ' make copy
  IF tmpB2.6 = 1 THEN               ' in 2nd or 4th quadrant?
    tmpB1 = 0 - tmpB1               ' yes, move to 1st/3rd
  ENDIF
  tmpB1.7 = 0                       ' reduce to 1st quadrant
  READ SineTable + tmpB1, tmpB1     ' read sine
  IF tmpB2.7 = 1 THEN               ' was angle in 3rd/4th?
    tmpB1 = 0 - tmpB1               ' yes, adjust
  ENDIF
  RETURN tmpB1
ENDFUNC

' -----------------------------------------------------------------------

' Use: value = COS angle
' -- "value" returned as signed byte (-127 to 127)
' -- angle is expressed in Brads (0 - 255)

FUNC COS
  tmpB1 = __PARAM1 + $40            ' get angle (adjust phase)
  tmpB1 = SIN tmpB1                 ' call sine table
  RETURN tmpB1
ENDFUNC


' =======================================================================
' User Data
' =======================================================================

SineTable:
  DATA 000, 003, 006, 009, 012, 016, 019, 022
  DATA 025, 028, 031, 034, 037, 040, 043, 046
  DATA 049, 051, 054, 057, 060, 063, 065, 068
  DATA 071, 073, 076, 078, 081, 083, 085, 088
  DATA 090, 092, 094, 096, 098, 100, 102, 104
  DATA 106, 107, 109, 111, 112, 113, 115, 116
```

```
  DATA 117, 118, 120, 121, 122, 122, 123, 124
  DATA 125, 125, 126, 126, 126, 127, 127, 127
  DATA 127
```



RANDOM

RANDOM *Seed {, Duplicate}*

Function
Generate a pseudo-random number.

- ***Seed*** is a variable whose bits will be scrambled to produce a random number. *Seed* acts as **RANDOM**'s input value and its result output. Each pass through **RANDOM** stores the next number, in the pseudo-random sequence, in *Seed*.
- ***Duplicate*** is an optional variable that, if provided, will receive a copy of *Seed* after **RANDOM**. This variable may be modified without affecting the value of *Seed* for the **RANDOM** instruction. Note: *Duplicate* must be the same size (byte or word) as *Seed*.

Explanation
RANDOM generates pseudo-random numbers ranging from 0 to 255 (if *Seed*s a byte) or 0 to 65535 (if *Seed* is a word). The value is called "pseudo-random" because it appears random, but are generated by a logic operation that uses the initial value in *Seed* to "tap" into a sequence of essentially random numbers. If the same initial value, called the "seed", is always used, then the same sequence of numbers will be generated. The following example demonstrates this:

```
Start:
  OUTPUT RB                   ' make RB outputs (LEDs)

Main:
  DO
    result = 123              ' set initial "seed" value
    RANDOM result                  ' generate random number
    RB = result               ' show the result on RB
    PAUSE 100
  LOOP
```

In this example, the same number would appear on RB over and over again. This is because the same seed value was used each time; specifically, the first line of the loop sets result to 123. The **RANDOM** command really needs a different seed value each time. Moving the "result =" line out of the loop will solve this problem, as in:

```
Start:
  OUTPUT RB                   ' make RB outputs (LEDs)

Main:
  result = 123                     ' set initial "seed" value
  DO
    RANDOM result                  ' generate random number
    RB = result               ' show the result on RB
    PAUSE 100
  LOOP
```

Here, result is only initialized once, before the loop. Each time through the loop, the previous value of result, generated by **RANDOM**, is used as the next seed value. This generates a more desirable set of pseudo-random numbers.

In applications requiring more apparent randomness, it's necessary to "seed" **RANDOM** with a more random value every time. For instance, in the digital dice example program, **RANDOM** is executed continuously (using the previous resulting number as the next seed value) while the program waits for the user to press a button. Since the user can't control the timing of button presses very accurately, the results approach true randomness.

Related project: Digital Dice

RCTIME

Example

RCTIME *Pin, State, Variable {, Resolution}*

Function

Measure time while *Pin* remains in *State*; usually to measure the charge/discharge time of resistor/capacitor (RC) circuit.

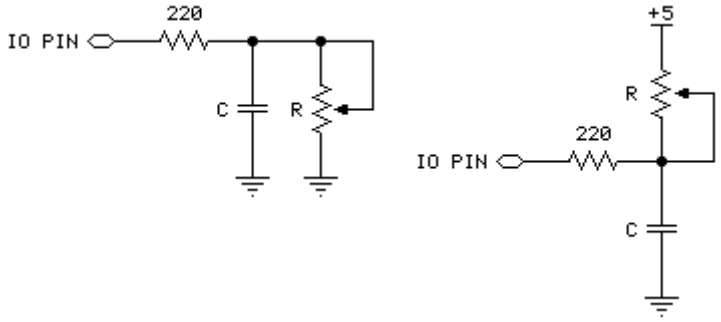
- ***Pin*** is any SX IO pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***State*** is a constant (0 - 1) that specifies the desired state to measure. Once *Pin* is not in *State*, the command ends and stores the result in *Variable*.
- ***Variable*** is a byte or word variable in which the time measurement will be stored. The unit of time for *Variable* is 2 microseconds multiplied by *Resolution* (if specified).
- ***Resolution*** is an optional variable or constant (1 - 255) that specifies the units for *Variable*, in increments of 2 microseconds (default is 1 when not specified).

Explanation

RCTIME can be used to measure the charge or discharge time of a resistor/capacitor circuit. This allows you to measure resistance or capacitance; use R or C sensors such as thermistors or capacitive humidity sensors or respond to user input through a potentiometer. In a broader sense, **RCTIME** can also serve as a fast, precise stopwatch for events of very short duration.

When **RCTIME** executes, it starts a counter. It stops this counter as soon as the specified pin is no longer in *State* (0 or 1). If pin is not in *State* when the instruction executes, **RCTIME** will return 0 in *Variable*. If pin remains in *State* longer than 255 timing cycles **RCTIME** returns 0.

The figure below shows suitable RC circuits for use with **RCTIME**. Circuit A is preferred, because the SX TTL logic threshold is approximately 1.4 volts. This means that the voltage seen by the pin will start at 5V then fall to 1.4V (a span of 3.6V) before **RCTIME** stops. With Circuit B, the voltage will start at 0V and rise to 1.4V (spanning only 1.4V) before **RCTIME** stops (this could be changed by setting the *Pin* threshold to CMOS). For the same combination of R and C, Circuit A will yield a higher count, and therefore more resolution than Circuit B.



(A) Use with State = 1 (B) Use with State = 0

Here's a typical sequence of instructions for Circuit A, using a 0.1 µF capacitor and a 10 kΩ pot.

```
Start:
  OUTPUT RB                    ' make LED pins outputs

Main:
  HIGH RA.0                    ' charge the capacitor
  PAUSEUS 250                  ' for 250 us
  RCTIME RA.0, 1, analog, 5    ' measure in 10 us (2 x 5) units
  RB = analog
  PAUSE 100
  GOTO Main
```

Using **RCTIME** is very straightforward, except for one detail: For a given R and C, what value will **RCTIME** return? It's easy to figure, based on a value called the RC time constant, or tau (τ) for short. Tau represents the time required for a given RC combination to charge or discharge by 63 percent of the total change in voltage that they will undergo. More importantly, the value t is used in the generalized RC timing calculation. Tau's formula is just R multiplied by C:

τ = R x C

The general RC timing formula uses τ to tell us the time required for an RC circuit to change from one voltage to another:

time = $-\tau \times (\ln(V_{final} / V_{initial}))$

In this formula ln is the natural logarithm; it's a key on most scientific calculators. Let's do some math. Assume we're interested in a 10 kΩ resistor and 0.1 µF capacitor. Calculate τ:

$\tau = (10 \times 10^{3}) \times (0.1 \times 10^{-6}) = 1 \times 10^{-3}$

The RC time constant is 1×10^{-3} or 1 millisecond. Now calculate the time required for this RC circuit to go from 5V to 1.4V (as in Circuit A):

Time = $-1 \times 10^{-3} \times (\ln(5.0v \div 1.4v)) = 1.273 \times 10^{-3}$

Using SX/B the unit of time is 2 µs, that time (1.273×10^{-3}) works out to about 635 units -- which exceeds the default resolution of **RCTIME**. What we can do is divide by 635 by 255 (byte value limit) to determine the smallest *Resolution* required to support the RC combination. In this case it works out to 2.49, so setting *Resolution* to 3 will allow us to measure the RC network with the greatest accuracy; in this case our measurement units will now be six microseconds (2 x 3). Note that setting *Resolution* too low will result in **RCTIME** returning zero.

Another handy rule of thumb can help you calculate how long to charge/discharge the capacitor before **RCTIME**. In the example above that's the purpose of the **HIGH** and **PAUSE** commands. A given RC charges or discharges 98 percent of the way in five time constants (5 x R x C). In Circuits A and B, the charge/discharge current passes through the 220 Ω series resistor and the capacitor. So if the capacitor were 0.1 µF, the minimum charge/discharge time should be:

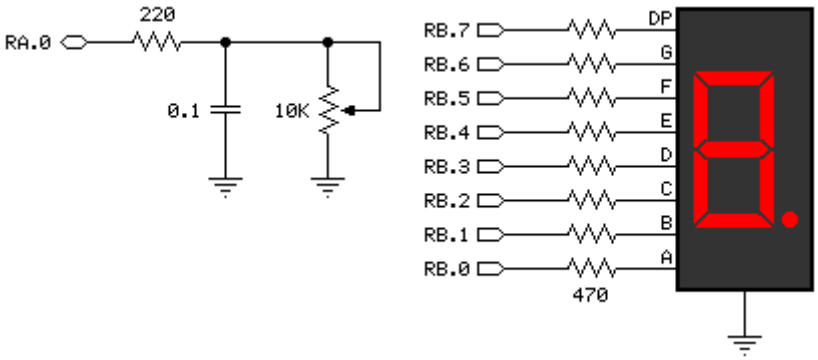
Charge time = $5 \times 220 \times (0.1 \times 10^{-6}) = 110 \times 10^{-6}$

So it takes only 110 µs for the capacitor to charge/discharge, meaning that the 250 microsecond charge/discharge time of the example is plenty.

A final note about the circuits above: You may be wondering why the 220 Ω resistor is necessary at all. Consider what would happen if resistor R was a potentiometer, and were adjusted to 0 Ω. When the I/O pin went high to discharge the capacitor, it would see a short direct to ground. The 220 Ω series resistor would limit the short circuit current to 5V ÷ 220 Ω = 23 mA and protect the SX IO pin from damage. (Actual current would be quite a bit less due to internal resistance of the pin's output driver, but you get the idea).

RCTIME Example

Syntax




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Reads a POT using RCTIME and converts to a decimal digit, 0 - 9.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "RCTIME"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

PotPin            PIN   RA.0              ' IO pin for RCTIME
Display           PIN   RB OUTPUT               ' 7-segment LED
' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

analog            VAR   Word             ' pot value
WATCH analog

' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  Display = %00000000                     ' clear display

Main:
  HIGH PotPin                             ' charge capacitor
```

```
  PAUSEUS 250                             ' for 250 usecs
  RCTIME PotPin, 1, analog                ' read pot (2 us units)
  analog = analog / 50                    ' convert to 0 - 9
  READ SegMap + analog_LSB, Display       ' put digit into display
  PAUSE 100                         ' wait 0.1 secs
  GOTO Main

' ===========================================================================
' User Data
' ===========================================================================

SegMap:                             ' segments maps
'         .gfedcba
  DATA      %00111111                     ' 0
  DATA      %00000110                     ' 1
  DATA      %01011011                     ' 2
  DATA      %01001111                     ' 3
  DATA      %01100110                     ' 4
  DATA      %01101101                     ' 5
  DATA      %01111101                     ' 6
  DATA      %00000111                     ' 7
  DATA      %01111111                     ' 8
  DATA      %01100111                     ' 9
```

READ | READINC

Examples

READ | **READINC***Base {+ Offset}, Variable {, Variable, ...}*

Function
Read one or more bytes from a table.

- ***Base*** is the base address of data to be read which may be specified as a **DATA** or **WDATA** statement label, or as a string-pointer variable created by the compiler (see below).
- ***Offset*** is an optional variable indicating the relative position (to *Base*) for the **READ** operation. This may be specified when *Base* is a label, or can be created by the compiler when strings are used (see below).
- ***Variable*** is a byte or word variable.

Explanation
The **DATA** directive can be used to create [read-only] tables for SX/B programs. The **READ** instruction is used to move one or more table values into the specified byte variable(s)

If **READINC** is used, the *Offset* wil be automatically incremented to to point to the next **DATA** item. If *Offset* is not used the **Base** will be incremented.

```
Start:
  OUTPUT RB

Main:
  FOR idx = 0 TO 3
    READ Pattern + idx, RB                ' move pattern to LEDs
    PAUSE 100
  NEXT idx
  FOR idx = 4 TO 1 STEP -1
    READ Pattern + idx, RB
    PAUSE 100
    NEXT idx
  GOTO Main                               ' do it again

' -----------------------------------------------------------------------------

Pattern: ' LED patterns
  DATA %00000000
  DATA %00011000
  DATA %00111100
  DATA %01111110
  DATA %11111111
```

As of version 1.4, SX/B can handle inline strings (of two or more characters) and z-strings (of any length) stored in **DATA** statements. Using **READ**, the following subroutine will send a string to a defined serial port:

```
' Use: TX_STR [ string | label ]
' -- "string" is an embedded literal string
' -- "label" is DATA statement label for stored z-String

SUB TX_STR
  tmpW1 = __WPARAM12                    ' get offset/base
  DO
    READ tmpW1, tmpW3                   ' read a character
    IF tmpW3 = 0 THEN EXIT              ' if 0, string complete
    SEROUT SOut, Baud, tmpW3            ' send the byte
    INC tmpW1                           ' point to next character
  LOOP
ENDSUB
```

This subroutine expects two parameters: the base address and the character offset of the string. The subroutine uses a word variable to accept the base+offset string pointer that is passed as a parameter. When a literal string or **DATA** label is specified as the TX_STR parameter, the compiler inserts the appropriate values that form a pointer to the string. Using the subroutine above strings can be transmitted like this:

```
TX_STR SUB 2                ' strings use two parameters

' -----------------------------------------------------------------------

Main:
  TX_STR "Version "         ' inline string (compiles to z-string)
  TX_STR VerNum
  END

' -----------------------------------------------------------------------

VerNum:
  DATA "1.0", 0         ' defined z-string
```

Note that when using a label as a subroutine parameter it must be defined before use, and the SX/B compiler adds the terminating zero to inline strings when there are two are more characters. If the following syntax is used:

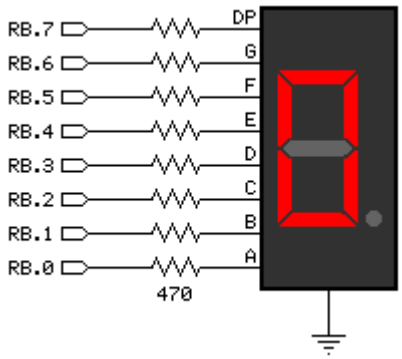
```
 TX_STR "X" ' character passed by value
```

an error will be raised as single characters are passed by value (one parameter), not by string pointer reference (two parameters). The solution is to create a subroutine for sending a single character that is also used by the TX_STR subroutine. See the examples page for details.

Related instructions: DATA/WDATA and LOOKUP

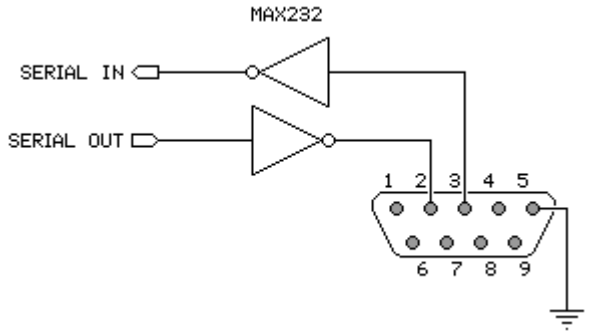
READ Examples

Syntax




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Uses READ to move values from a DATA table to a variable. In this
' program, the table data holds segment patterns for a 7-segment LED.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "READ"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

LEDs        PIN   RB OUTPUT   ' 7-segment display


' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

Dash CON %01000000

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

value             VAR   Byte        ' value to display
tmpB1             VAR   Byte        ' subroutine work var

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
```

```
' -----------------------------------------------------------------------

PUT_DIGIT   FUNC  1,1

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  TRIS_LEDs = %00000000             ' make LEDs outputs

Main:
  DO
  FOR value = 0 TO 16               ' demo values (last invalid)
  LEDs = PUT_DIGIT value            ' update the display
  PAUSE 500                         ' pause 1/2 second
  NEXT
  LOOP

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' Use: destination = PUT_DIGIT value
' -- converts number in 'value' to 7-segment digit pattern
' and places it in 'destination'

FUNC PUT_DIGIT
  tmpB1 = __PARAM1                  ' copy value
  IF tmpB1 <= $F THEN               ' check range
     READ SegMap + tmpB1, tmpB1     ' read table value
  ELSE
    tmpB1 = Dash              ' display dash
  ENDIF
  RETURN tmpB1
ENDFUNC

' =======================================================================
' User Data
' =======================================================================

SegMap:                         ' segments maps
'       .gfedcba
  DATA %00111111              ' 0
  DATA %00000110              ' 1
  DATA %01011011              ' 2
  DATA %01001111              ' 3
  DATA %01100110              ' 4
  DATA %01101101              ' 5
  DATA %01111101              ' 6
  DATA %00000111              ' 7
  DATA %01111111              ' 8
  DATA %01100111              ' 9
  DATA %01110111              ' A
  DATA %01111100              ' b
  DATA %00111001              ' C
  DATA %01011110              ' d
  DATA %01111001              ' E
  DATA %01110001              ' F
```




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' This program demonstrates the use of strings in an SX/B program. As of
' version 1.4, the SX/B READ instruction can accept a variable base and
' offset. These values may be created by the compiler by specifying the
' label of a stored z-string, or by using an inline string constant.
'
' The subroutine TX_STR accepts the base and offset values of a stored or
' inline string and will transmit them to a connected terminal -- the
' construction of the subroutine allows the string to cross SX page
' boundaries.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "READ_STR"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

SOut        PIN   RA.0 OUTPUT       ' serial output

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Baud        CON   "T9600"           ' use with MAX232/USB2SER
CR          CON   13                ' carriage return
LF          CON   10                ' line feed

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

tmpB1             VAR   Byte              ' subroutine work vars
tmpB2             VAR   Byte
tmpB3             VAR   Byte
tmpB4             VAR   Byte
tmpW1             VAR   Word
```

```
' ================================================================================
 PROGRAM Start
' ================================================================================

' --------------------------------------------------------------------------------
' Subroutine Declarations
' --------------------------------------------------------------------------------

TX_BYTE     SUB   1, 2              ' transmit byte
TX_STR            SUB   2, 4              ' transmit string

' --------------------------------------------------------------------------------
' Program Code
' --------------------------------------------------------------------------------

Start:
  PLP_A = %0001                     ' pull-up unused pins
  PLP_B = %00000000
  PLP_C = %00000000

Main:
  TX_STR "SX/B makes string output easy!"       ' send inline string
  TX_STR CrLf                               ' send stored z-string
  TX_STR TestStr, 14                        ' sub-string
  TX_STR CrLf
  TX_STR TestStr, 13, 16                    ' sub-string at offset
  TX_STR CrLf
  TX_STR TestStr, 8, 31
  TX_STR Version
  TX_STR CrLf
  TX_BYTE LF, 2
  PAUSE 1000
  GOTO Main

' --------------------------------------------------------------------------------
' Subroutine Code
' --------------------------------------------------------------------------------

' Use: TX_BYTE theByte {, count}
' -- transmit "theByte" at "Baud" on "SOut"
' -- optional "count" may be specified (must be > 0)

SUB TX_BYTE
  tmpB1 = __PARAM1                          ' save byte
  IF __PARAMCNT = 1 THEN                    ' if no count
    tmpB2 = 1                               ' set to 1
  ELSE                                      ' otherwise
    tmpB2 = __PARAM2                        ' get count
  IF tmpB2 = 0 THEN                         ' do not allow 0
    tmpB2 = 1
  ENDIF
  ENDIF
  DO WHILE tmpB2 > 0                        ' loop through count
    SEROUT SOut, Baud, tmpB1                ' send the byte
    DEC tmpB2                               ' decrement count
  LOOP
ENDSUB
' --------------------------------------------------------------------------------

' Use: TX_STR [string | label] {, count {, offset }}
```

```
' -- "string" is an embedded string constant
' -- "label" is DATA statement label for stored z-String
' -- "count" is number of characters to send (0 is entire string)
' -- "offset" is offset from head of string (0 is head)

SUB TX_STR
  tmpW1 = __WPARAM12                    ' get offset+base
  tmpB3 = 0 ' do whole string
  IF __PARAMCNT >= 3 THEN               ' count specified?
    tmpB3 = __PARAM3                    ' -- yes, get count
  ENDIF
  IF __PARAMCNT = 4 THEN                ' offset specified?
    tmpW1 = tmpW1 + __PARAM4            ' -- yes, update it
  ENDIF
  DO
    READ tmpW1, tmpB4                   ' read a character
    IF tmpB4 = 0 THEN EXIT              ' if 0, string complete
    TX_BYTE tmpB4                       ' send character
    DEC tmpB3                           ' update count
    IF tmpB3 = 0 THEN EXIT              ' terminate if done
    INC tmpW1                           ' point to next character
  LOOP
ENDSUB

' ===========================================================================
' User Data
' ===========================================================================

TestStr:
  DATA "Parallax, Inc. SX/B Compiler Version 1.50", 0

Version:
  DATA "1.50", 0

CrLf:
  DATA CR, LF, 0
```

RESETWDT

Example

RESETWDT

Function
Resets the SX Watchdog Timer.

Explanation
When the watchdog timer is enabled (through the **DEVICE** directive), the **RESETWDT** instruction must be used periodically to clear the timer in order prevent a device reset. If the program becomes stuck and **RESETWDT** instruction does not get called in time the watchdog timer will reset the SX.

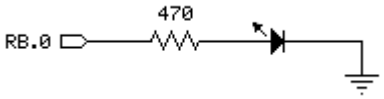
An internal timer controls the watchdog timeout. By setting bit 3 (PSA) of the **OPTION** register, bits 2..0 (PS2..PS0) can be used to set the watchdog timeout period.

PS2	PS1	PS0	Scale	Period
0	0	0	1:1	18 ms
0	0	1	1:2	37 ms
0	1	0	1:4	73 ms
0	1	1	1:8	146 ms
1	0	0	1:16	293 ms
1	0	1	1:32	585 ms
1	1	0	1:64	1.2 s
1	1	1	1:128	2.3 s

Reated instructions: END and SLEEP

RESETWDT Example

Syntax




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Demonstrates the use of the RESETWDT instruction in SX/B. When the
' RESETWDT instruction is removed (with comment), the LED will flash due
' to the SX being reset after the watchdog timer timeout.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX, WATCHDOG
FREQ        4_000_000
ID          "RESETWDT"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

Led         PIN   RB.0 OUTPUT

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  OPTION = %11111110                    ' timeout = ~1.2 seconds
  LOW Led                               ' LED off
  PAUSE 500                             ' wait for LED on

Main:
  DO
    ' RESETWDT                           ' prevent timeout
    HIGH Led                            ' LED on
  LOOP
```

REVERSE

Example

REVERSE *Pin*

Function
Reverse the data direction register (TRIS) bit of the specified *pin*.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).

Explanation
REVERSE is convenient way to switch the I/O direction of a pin. If the pin is an input, **REVERSE** makes it an output; if it's an output, **REVERSE**makes it an input.

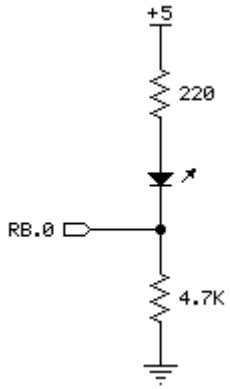
Remember that "input" really has two meanings: (1) Setting a pin to input makes it possible to check the state (1 or 0) of external circuitry connected to that pin. The current state is in the corresponding bit of the associated port register. (2) Setting a pin to input also disconnects the output driver, possibly affecting circuitry being controlled by the pin.

```
HIGH RA.3                   ' RA.3 is output and high (1)
PAUSE 100
REVERSE RA.3                ' RA.3 is now an input
```

Related instructions: INPUT and OUTPUT

REVERSE Example

Syntax




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Allows an LED to have one of two brighness levels by controlling port
' pin with REVERSE. When the LED pin is an input (Hi-Z), the current
' passes through both resistors, causing the LED to be dim. When the
' LED pin is an output low, the current only has to pass through one
' resistor, hence the LED is brighter.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "REVERSE"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

Led         PIN   RB.0 OUTPUT

' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  Led = 0                           ' put low in output bit

Main:
  DO
    PAUSE 250                       ' wait 1/4 second
    REVERSE Led                     ' change brightness
  LOOP
```

SCHMITT (SX48/52 Only)

SCHMITT *Pin {, Enable}*

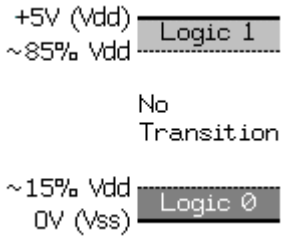
Function

Configures the internal Schmitt trigger for *Pin* on the SX48 or SX52. This command does not apply to the SX18, SX20, or SX28 (use the ST_B and ST_C registers).

- ***Pin*** is any SX48/52 I/O pin (RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***Enable*** is a constant, 0 or 1, that enables (1) or disables (0) the Schmitt trigger of *Pin*. When not specified, *Enable* defaults to 1.

Explanation

Every I/O pin in port B through port E can be set to normal or Schmitt-Trigger input. This can be configured by writing to the appropriate Schmitt-Trigger register (ST_B, ST_C, ST_D and ST_E). The I/O pins are set to normal input mode by default. Schmitt-Trigger mode can be activated for all pins, regardless of pin direction but really matter only when the associated pin is set to input mode. By configuring Schmitt-Trigger mode on input pins, the SX chip can be less sensitive to minor noise on the input pins. The figure below details the characteristics of Schmitt-Trigger inputs.



Schmitt Trigger Characteristics

Schmitt-Trigger inputs are conditioned with a large area of hysteresis. The threshold for a logic 0 is 15% of Vdd and the threshold for a logic 1 is 85% of Vdd. The input pin defaults to an unknown state until the initial voltage crosses a logic 0 or logic 1 boundary. A voltage must cross above 85% of Vdd to be interpreted as a logic 1 and must cross below 15% of Vdd to be interpreted as a logic 0. If the voltage transitions somewhere between the two thresholds, the interpreted logic state remains the same as the previous state.

```
Start:
  SCHMITT RB.0, 1                  ' enable Schmitt trigger
  SCHMITT RB.1                     ' enable Schmitt trigger
  SCHMITT RB.2, 0                  ' disable Schmitt trigger
```

Related instructions: CMOS, TTL, and PULLUP

SERIN

Example

SERIN *Pin, BaudMode, ByteVar {, Timeout, ToLabel}*

Function

Receive asynchronous serial byte (e.g., RS-232).

- ***Pin*** is any SX IO pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***BaudMode*** is a string constant that specifies serial timing and configuration. SX/B will raise an error if the baud rate specified exceeds the ability of the target **FREQ**.
- ***ByteVar*** is a variable in which the serial data byte is stored.
- ***Timeout*** is an optional byte or word variable/constant that tells **SERIN** how long to wait (in milliseconds) for incoming data. If data does not arrive in time, the program will jump to the address specified by *ToLabel*.
- ***ToLabel*** is an optional label that must be provided along with *Timeout*, indicating where the program should go in the event that data does not arrive within the period specified by *Timeout*.

Quick Facts

	FREQ = 4 MHz	FREQ = 20 MHz	FREQ = 50 MHz
Maximum Baud Rate †	38400	115200	> 230400
Baud Modes	T (true), N (inverted), OT (open, true), ON (open, inverted)		
Units in *Timeout*	1 millisecond		

† When used without Timeout parameter.

Explanation

Receive asynchronous serial byte at the selected baud rate and mode using no-parity, 8-data bits, and 1-stop bit. If *Timeout* is specified, jump to *ToLabel* if serial byte does not arrive within *Timeout* milliseconds.

One of the most popular forms of communication between electronic devices is serial communication. There are two major types of serial communication: asynchronous and synchronous. The **SERIN** and **SEROUT** commands are used to receive and send asynchronous serial data. See the SHIFTIN and SHIFTOUTcommand for information on the synchronous method.

The term asynchronous means "no clock." More specifically, "asynchronous serial communication" means data is transmitted and received without the use of a separate "clock" wire. Data can be sent using as little as two wires: one for data and one for ground.

This simple demo shows how to receive a single byte through RA.0 at 2400 baud, 8N1, inverted:

```
Main:
  DO
    SERIN RA.0, N2400, sData        ' wait for byte
    LEDs = sData                    ' put byte value on LEDs
  LOOP
```

Here, **SERIN** will wait for and receive a single byte of data through pin RA.0 and store it in the variable *sData*. If the SX were connected to a PC running a terminal program (set to the same baud rate) and the user pressed the "A" key on the keyboard, after the **SERIN** command executed, the variable *sData* would contain 65, the ASCII code for the letter "A".

When used without a *Timeout* parameter, **SERIN** block program operation until a byte arrives. Where this condition will have an adverse affect on program operation a *Timeout* parameter and label can be specified. In this update to the example above, the program will wait for two seconds (2000 milliseconds) for an incoming byte; if no byte arrives the program will be redirected to No_Char and the LEDs will be blanked.

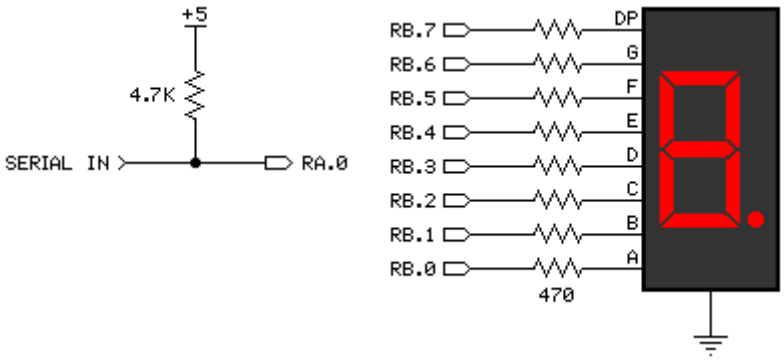
```
Main:
  DO
    SERIN RA.0, N2400, sData, 2000, No_Char     ' wait for byte
    LEDs = sData                                ' put byte value on LEDs
  LOOP

No_Char:
  LEDs = %00000000                              ' clear LEDs
  GOTO Main
```

Note: Interrupts will interfere with the proper operation of **SERIN** and, in most cases, should be disabled before the **SERIN** instruction is used. If the interrupt is short and designed to run the same number of cycles under any condition, the *EffectiveHz* parameter of FREQ may be used.

Related instruction: SEROUT
Related projects: Serial LCD and RFID Reader Interface

SERIN Example

Syntax




```
' =====================================================================
'
' File...... SERIN.SXB
' Purpose... Create simple 7-segment display controller (one digit)
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =====================================================================
' ---------------------------------------------------------------------
' Program Description
' ---------------------------------------------------------------------
'
' Waits for a command from a host controller and displays hex value or
' individual segments on a 7-segment LED display.
'
' Commands:
' "!LEDH", value -- display hex digit value (0 .. F)
' "!LEDS", value -- display individual segments

' ---------------------------------------------------------------------
' Device Settings
' ---------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "SERIN"

' ---------------------------------------------------------------------
' IO Pins
' ---------------------------------------------------------------------

Sio         PIN   RA.0 INPUT        ' pull-up via 4.7K
LEDs        PIN   RB OUTPUT         ' 7-segment display
' ---------------------------------------------------------------------
' Constants
' ---------------------------------------------------------------------

Baud        CON   "OT9600"
```

```
Dash        CON   %01000000

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

serByte     VAR   Byte              ' serial I/O byte
tmpB1             VAR   Byte              ' parameter

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

RX_BYTE     FUNC  1                 ' receive serial byte
MAKE_DIGIT  FUNC  1                 ' make 7-segs digit

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  PLP_A = %0001                     ' pull up unused pins
  PLP_C = %00000000
  LEDs = %00000000                       ' clear display


Main: ' wait for header
  serByte = RX_BYTE
  IF serByte <> "!" THEN Main
  serByte = RX_BYTE
  IF serByte <> "L" THEN Main
  serByte = RX_BYTE
  IF serByte <> "E" THEN Main
  serByte = RX_BYTE
  IF serByte <> "D" THEN Main

Check_Hex:
  serByte = RX_BYTE
  IF serByte = "H" THEN                  ' hex display?
    serByte = RX_BYTE                    ' get value
    LEDs = MAKE_DIGIT serByte
    GOTO Main
  ENDIF

Check_Segments:
  IF serByte = "S" THEN                  ' segments display?
    LEDs = RX_BYTE                       ' get segments
    GOTO Main
  ENDIF

Check_X:                            ' add features here
  ' for future use
  GOTO Main

' -----------------------------------------------------------------------
' Subroutine Code
```

```
' ------------------------------------------------------------------------

' Use: rxInput = RX_BYTE
' -- reads byte from serial input and places it in 'rxInput'

FUNC RX_BYTE
  SERIN Sio, Baud, tmpB1                  ' receive a byte
  RETURN tmpB1
ENDFUNC

' ------------------------------------------------------------------------

' Use: destination = MAKE_DIGIT value
' -- converts number in 'value' to 7-segment digit pattern
' and places it in 'destination'

FUNC MAKE_DIGIT
  tmpB1 = __PARAM1                        ' copy value
  IF tmpB1 <= $F THEN                     ' check range
    READ SegMap + tmpB1, tmpB1                 ' read table value
  ELSE
    tmpB1 = Dash                      ' display dash
  ENDIF
  RETURN tmpB1
ENDFUNC

' ========================================================================
' User Data
' ========================================================================

SegMap: ' segments maps
'       .gfedcba
  DATA %00111111                     ' 0
  DATA %00000110                     ' 1
  DATA %01011011                     ' 2
  DATA %01001111                     ' 3
  DATA %01100110                     ' 4
  DATA %01101101                     ' 5
  DATA %01111101                     ' 6
  DATA %00000111                     ' 7
  DATA %01111111                     ' 8
  DATA %01100111                     ' 9
  DATA %01110111                     ' A
  DATA %01111100                     ' b
  DATA %00111001                     ' C
  DATA %01011110                     ' d
  DATA %01111001                     ' E
  DATA %01110001                     ' F
```

The following BASIC Stamp 2 program can be used to to demonstrate the LED display.

```
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Program Description ]---------------------------------------------
'
' Test program for SERIN.SXB
```

```
' -----[ I/O Definitions ]-------------------------------------------------

Sio         PIN   15

' -----[ Constants ]-------------------------------------------------------

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    T1200 CON 813
    T2400 CON 396
    T4800 CON 188
    T9600 CON 84
    T19K2 CON 32
    TMidi CON 12
    T38K4 CON 6
  #CASE BS2SX, BS2P
    T1200 CON 2063
    T2400 CON 1021
    T4800 CON 500
    T9600 CON 240
    T19K2 CON 110
    TMidi CON 60
    T38K4 CON 45
  #CASE BS2PX
    T1200 CON 3313
    T2400 CON 1646
    T4800 CON 813
    T9600 CON 396
    T19K2 CON 188
    TMidi CON 108
    T38K4 CON 84
#ENDSELECT

SevenBit    CON   $2000
Inverted    CON   $4000
Open        CON   $8000
Baud        CON   Open + T9600

' -----[ Variables ]-------------------------------------------------------

idx         VAR   Nib
segments    VAR   Byte
value             VAR   Nib

' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    FOR idx = 0 TO 5
      segments = (1 << idx)
      SEROUT Sio, Baud, ["!LEDS", segments]
      PAUSE 50
    NEXT
    SEROUT Sio, Baud, ["!LEDH", value]
    PAUSE 500
    value = value + 1 // 16
  LOOP
  END
```

SEROUT

Example

SEROUT *Pin, BaudMode, Value*

Function

Transmit asynchronous serial byte (e.g., RS-232).

- ***Pin*** is any SX IO pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***BaudMode*** is a string constant that specifies serial timing and configuration. SX/B will raise an error if the baud rate specified exceeds the ability of the target **FREQ**.
- ***Value*** is a byte variable or constant (0 - 255) to be transmitted.

Quick Facts

	FREQ = 4 MHz	FREQ = 20 MHz	FREQ = 50 MHz
Maximum Baud Rate	57600	460800	> 921600
Baud Modes	T (true), N (inverted), OT (open, true), ON (open, inverted)		

Explanation

Transmit asynchronous serial byte at the selected baud rate and mode using no-parity, 8-data bits, and 1-stop bit.

```
SEROUT RA.0, T9600, "A"
```

In the example above, the SX will transmit the letter "A" (decimal 65) to an external device at 9600 baud, in true mode on pin RA.0. Since **SEROUT** requires a substantial amount of assembly code a good way to save program space is by placing **SEROUT** in a subroutine.

For example:

```
TX_CHAR:
  temp1 = __PARAM1                  ' save byte to send
  temp2 = __PARAM2                  ' save number of repeats
  DO WHILE temp2 > 0
    SEROUT Sio, Baud, temp1         ' send the byte
    DEC temp2                       ' update count
  LOOP
  RETURN
```

This subroutine takes two parameters: the first is the byte to transmit and the second is the number of times to transmit that byte. By using the second parameter sending "**********" is as easy as (when the subroutine is declared with **SUB**):

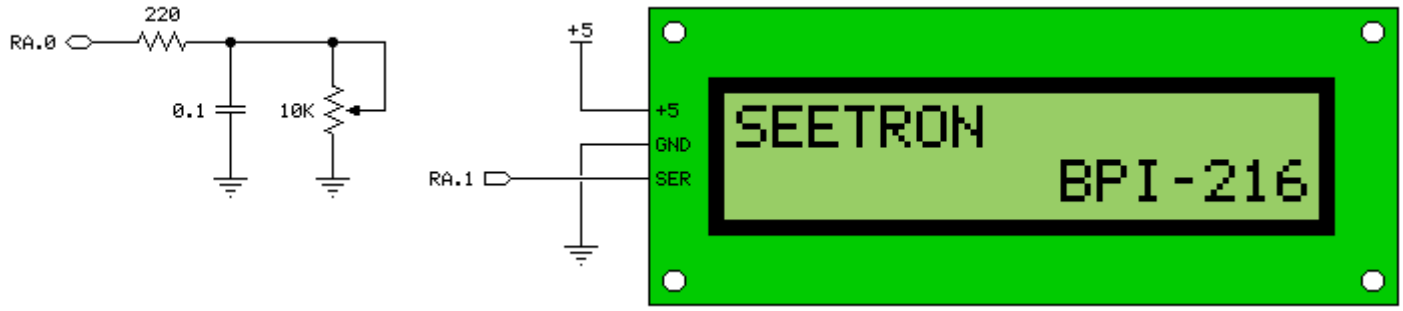
```
TX_CHAR "*", 10 ' send "**********"
```

Note: Interrupts will interfere with the proper operation of **SEROUT** and, in most cases, should be disabled before the **SEROUT** instruction is used. If the interrupt is short and designed to run the same number of cycles under any condition, the *EffectiveHz* parameter of FREQ may be used.

Related instruction: SERIN
Related projects: Serial LCD and RFID Reader Interface

SEROUT Example

SEROUT




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Reads a potentiometer with RCTIME and sends the value to a SEETRON
' (www.seetron.com) serial LCD.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "SEROUT"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

PotPin            PIN   RA.0              ' I/O pin for RCTIME
Sout        PIN   RA.1 OUTPUT       ' output to SEETRON 2x16

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Baud        CON   "N9600"

LcdI        CON   $FE               ' instruction
LcdCls            CON   $01                 ' clear the LCD
LcdHome     CON   $02               ' move cursor home
LcdCrsrL    CON   $10               ' move cursor left
LcdCrsrR    CON   $14               ' move cursor right
LcdDispL    CON   $18               ' shift chars left
LcdDispR    CON   $1C               ' shift chars right
LcdDDRam    CON   $80               ' Display Data RAM control
LcdCGRam    CON   $40               ' Character Generator RAM
LcdLine1    CON   $80               ' DDRAM address of line 1
LcdLine2    CON   $C0               ' DDRAM address of line 2
```

```
' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

analog            VAR   Byte              ' pot value
char        VAR   Byte              ' character to send
idx         VAR   Byte              ' loop counter

line1             VAR   Byte(16)          ' line 1 buffer
line2             VAR   Byte(16)          ' line 2 buffer

tmpB1             VAR   Byte              ' subroutine work vars
tmpB2             VAR   Byte
tmpW1             VAR   Word

' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Subroutine DECLARATIONS
' -----------------------------------------------------------------------------

TX_BYTE     SUB   1, 2              ' transmit a byte
DELAY_US    SUB   1, 2              ' delay in microseconds
DELAY             SUB   1, 2              ' delay in milliseconds
UPDATE_L1   SUB   0                 ' update line 1 of LCD
UPDATE_L2   SUB   0                 ' update line 2 of LCD
UPDATE_LCD  SUB   0                 ' update both lines

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  DELAY 750                             ' let LCD initialize
  TX_BYTE LcdI                          ' clear screen, home cursor
  TX_BYTE LcdCls
  DELAY 1
  PUT line1, "POT: "                    ' initialize LCD buffer
  PUT line2, " "

Main:
  HIGH PotPin                           ' charge capacitor
  DELAY_US 250                          ' for 250 usecs
  RCTIME PotPin, 1, analog, 5                ' read pot (10 us units)

Show_Pot:
  tmpB1 = analog / 100                  ' get 100s value
  tmpB2 = __REMAINDER                   ' save 10s and 1s
  tmpB1 = tmpB1 + "0"                   ' convert 100s to ASCII
  PUT line1(5), tmpB1                   ' move to LCD buffer
  tmpB1 = tmpB2 / 10                    ' get 10s value
  tmpB2 = __REMAINDER                   ' save 1s
  tmpB1 = tmpB1 + "0"                   ' convert 10s to ASCII
  PUT line1(6), tmpB1                   ' move to LCD buffer
  tmpB1 = tmpB2 + "0"                   ' convert 1s to ASCII
  PUT line1(7), tmpB1                   ' move to LCD buffer
  UPDATE_L1                             ' update LCD
  DELAY 100                             ' wait 100 ms
```

```
   GOTO Main                            ' do it over

' -----------------------------------------------------------------------
' Subroutines Code
' -----------------------------------------------------------------------

' Use: TX_BYTE theByte {, repeats }
' -- first parameter is byte to transmit
' -- second (optional) parameter is number of times to send

SUB TX_BYTE
  tmpB1 = __PARAM1                      ' char to send
  IF __PARAMCNT = 1 THEN                ' if no repeats specified
    tmpB2 = 1                           ' - set to 1
  ELSE
    tmpB2 = __PARAM2                    ' - save repeats
  ENDIF
  DO WHILE tmpB2 > 0
    SEROUT Sout, Baud, tmpB1            ' send the character
    DEC tmpB2
  LOOP
ENDSUB

' -----------------------------------------------------------------------

' Use: DELAY_US us
' -- 'us' is delay in microseconds, 1 - 65535

SUB DELAY_US
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                    ' save byte value
  ELSE
    tmpW1 = __WPARAM12                  ' save word value
  ENDIF
  PAUSEUS tmpW1
ENDSUB

' -----------------------------------------------------------------------

' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535
SUB DELAY:
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                    ' save byte value
  ELSE
    tmpW1 = __WPARAM12                  ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB

' -----------------------------------------------------------------------

' Transfers line 1 buffer to LCD
' -- makes no change in LCD screen position

SUB UPDATE_L1
  TX_BYTE LcdI                          ' cursor to line 1, col 0
  TX_BYTE LcdLine1
  DELAY 1
  FOR idx = 0 TO 15
```

```
    TX_BYTE line1(idx)                    ' transfer buffer
  NEXT
ENDSUB

' -----------------------------------------------------------------------

' Transfers line 2 buffer to LCD
' -- makes no change in LCD screen position

SUB UPDATE_L2
  TX_BYTE LcdI                            ' cursor to line 2, col 0
  TX_BYTE LcdLine2
  DELAY 1
  FOR idx = 0 TO 15
    TX_BYTE line2(idx)                    ' transfer buffer
  NEXT
ENDSUB

' -----------------------------------------------------------------------

' Updates the LCD with both line buffers

SUB UPDATE_LCD
  TX_BYTE LcdI
  TX_BYTE LcdHome
  DELAY 1
  UPDATE_L1
  UPDATE_L2
ENDSUB
```

SHIFTIN

Example

SHIFTIN *DPin, CPin, ShiftMode, ByteVar {\Count} {, SpeedMult}*

Function
Shift data out to a synchronous serial device.

- ***DPin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7). This pin will be set to input mode.
- ***CPin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7). This pin will be set to output mode.
- ***ShiftMode*** is one of four predefined symbols that tells **SHIFTIN** the order in which data bits are to be arranged and the relationship of clock pulses to valid data. See the table below for value and symbol definitions.
- ***ByteVar*** is a variable which will hold the incoming data.
- ***Count*** is an optional constant (1 - 8) specifying how many bits are to be input by **SHIFTIN**. If no *Count* parameter is given **SHIFTIN** defaults to eight bits.
- ***SpeedMult*** is an optional constant that may be used to multiply the clock speed of **SHIFTIN** (with the limits of the current **FREQ** setting). When not specified the value of *SpeedMult*is set to 1.

Explanation
SHIFTIN and **SHIFTOUT** provide an easy method of connecting to synchronous serial devices. Synchronous serial differs from asynchronous serial (like **SERIN** and **SEROUT**) in that the timing of data bits (on a data line) is specified in relationship to clock pulses (on a clock line). Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is commonly used by controller peripherals like ADCs, DACs, clocks, memory devices, etc.

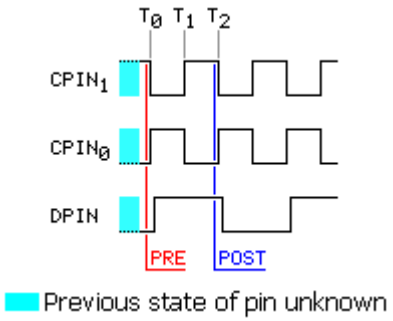
At their heart, synchronous-serial devices are essentially shift-registers; trains of flip-flops that pass data bits along in a bucket brigade fashion to a single data output pin. Another bit is output each time the appropriate edge (rising or falling, depending on the device) appears on the clock line.

The **SHIFTIN** instruction first forces the clock pin (*CPin*) to output mode and the data pin (*DPin*) to input mode. Then, **SHIFTIN**either reads the data pin and generates a clock pulse (PRE mode) or generates a clock pulse then reads the data pin (POST mode). Clock pulses are generated by inverting the state of *CPin*, allowing the programmer to determine *CPin* behavior by presetting the pin to the opposite state prior to the call. **SHIFTIN** continues to generate clock pulses and read the data pin for as many data bits as are required.

Making **SHIFTIN** work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data. Items to look for include: 1) which bit of the data arrives first; most significant bit (MSB) or least significant bit (LSB) and 2) is the first data bit ready before the first clock pulse (PRE) or before the second clock pulse (POST). The table below shows the values and symbols available for the *ShiftMode* parameter.

Symbol	Value	Meaning
MSBPRE	0	Data is MSB-first; sample bits before clock pulse
LSBPRE	1	Data is lsb-first; sample bits before clock pulse
MSBPOST	2	Data is MSB-first; sample bits after clock pulse
LSBPOST	3	Data is LSB-first; sample bits after clock pulse

SHIFTIN Timing



T_0 - T_1, T_1 - T_2, ...	~ 6 µs
Transmission Rate	~ 83 kBits/Sec

Here is a simple example:

```
SHIFTIN RC.0, RC.1, MSBPRE, result
```

Here, the **SHIFTIN** instruction will read pin RC.0 (*Dpin*) and will generate a clock signal on RA.1 (*Cpin*). The data that arrives on the *Dpin* depends on the device connected to it. Let's say, for example, that a shift register is connected and has a value of $AF (%10101111) waiting to be sent. Additionally, let's assume that the shift register sends out the most significant bit first, and the first bit is on the *Dpin* before the first clock pulse (MSBPRE). The **SHIFTIN** instruction above will generate eight clock pulses and sample the *Dpin* eight times. Afterward, the *result* variable will contain the value $AF.

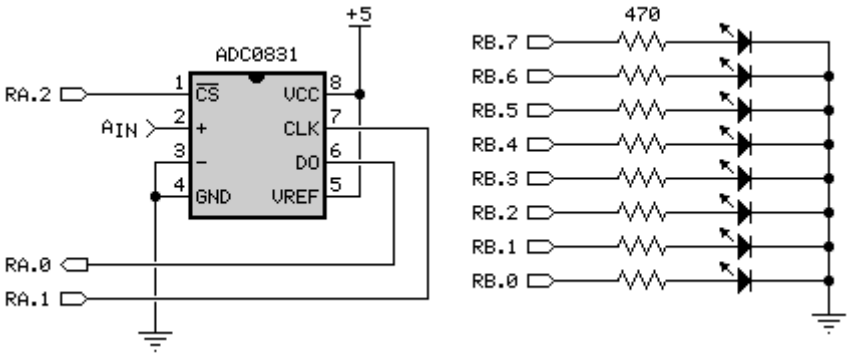
Some devices return more than eight bits. For example, the LTC1298 is a 12-bit ADC. To retrieve data from the LTC1298 would require two **SHIFTIN**calls to retrieve the 12-bit result.

```
 LOW CS
SHIFTOUT Dio, Clk, LSBFIRST, config\4          ' send config bits
SHIFTIN Dio, Clk, MSBPOST, resultHi\4          ' get data (upper nibble)
SHIFTIN Dio, Clk, MSBPOST, resultLo\8          ' get data (lower byte)
HIGH CS
```

Related instructions: SHIFTOUT
Related project: Thermometer

SHIFTIN Example

Syntax




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Reads value from an ADC0831 and places that value on LEDs connected to
' port RB.
'
' Note that the SpeedMult feature is used with SHIFTIN to bump the clock
' speed up to ~332 kBits/sec (ADC0831 max is 400).

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "SHIFTIN"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

Dpin        PIN   RA.0 INPUT        ' shift data
Cpin        PIN   RA.1 OUTPUT       ' shift clock
CS          PIN   RA.2 OUTPUT       ' chip select

LEDs        PIN   RB   OUTPUT


' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

tmpB1             VAR   Byte        ' parameter

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
```

```
' Subroutine Declarations
' -----------------------------------------------------------------------

GET_ADC     FUNC  1                 ' get value from ADC

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:

HIGH CS                             ' make CS output, no ADC
  LOW Cpin                          ' make clock 0-1-0

Main:
  LEDs = GET_ADC                    ' move ADC value to LEDs
  PAUSE 100                         ' wait 0.1 second
  GOTO Main

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

' Use: aVar = GET_ADC
' -- reads ADC0831 and places value into 'aVar'

FUNC GET_ADC
  CS = 0                            ' activate ADC0831
  SHIFTIN Dpin, Cpin, MSBPOST, tmpB1\1, 4       ' start conversion
  SHIFTIN Dpin, Cpin, MSBPOST, tmpB1, 4   ' shift in the data
  CS = 1                            ' deactivate ADC0831
  RETURN tmpB1
ENDFUNC
```

SHIFTOUT

Example

SHIFTOUT *DPin, CPin, ShiftMode, Value {|Count} {, SpeedMult}*

Function

Shift data out to a synchronous serial device.

- ***DPin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7). This pin will be set to output mode.
- ***CPin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7). This pin will be set to output mode.
- ***ShiftMode*** is one of two predefined symbols that tells **SHIFTOUT** the order in which data bits are to be arranged. See the table below for value and symbol definitions.
- ***Value*** is a variable or constant containing the data to be sent.
- ***Count*** is an optional constant (1 - 16) specifying how many bits are to be output by **SHIFTOUT**. If no *Count* parameter is given **SHIFTOUT** defaults to eight bits. When the *Count* parameter is given, the SX transmits the rightmost number of bits specified, regardless of the *ShiftMode*.
- ***SpeedMult*** is an optional constant that may be used to multiply the clock speed of **SHIFTOUT** (with the limits of the current **FREQ** setting). When not specified the value of *SpeedMult* is set to 1.

Explanation

SHIFTIN and **SHIFTOUT** provide an easy method of connecting to synchronous serial devices. Synchronous serial differs from asynchronous serial (like **SERIN** and **SEROUT**) in that the timing of data bits (on a data line) is specified in relationship to clock pulses (on a clock line). Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is commonly used by controller peripherals like ADCs, DACs, clocks, memory devices, etc.

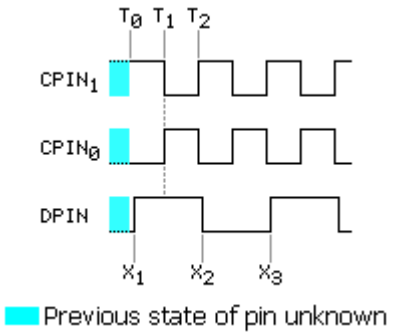
At their heart, synchronous-serial devices are essentially shift-registers; trains of flip-flops that pass data bits along in a bucket brigade fashion to a single data output pin. Another bit is output each time the appropriate edge (rising or falling, depending on the device) appears on the clock line.

The **SHIFTOUT** instruction first sets the clock and data pins to switch to output mode. Then, **SHIFTOUT** sets the data pin to the next bit state to be output and generates a clock pulse by inverting the state of the clock pin; this allows the programmer to set the desired clocking edge by presetting the clock pin to the opposite state prior to the call. **SHIFTOUT** continues to generate clock pulses and places the next data bit on the data pin for as many data bits as are required for transmission.

Making **SHIFTOUT** work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data. One of the most important items to look for is which bit of the data should be transmitted first; most significant bit (MSB) or least significant bit (LSB). The table below shows the values and symbols available for the *ShiftMode* parameter.

Symbol	Value	Meaning
LSBFIRST	0	Data is shifted out LSB-first
MSBFIRST	1	Data is shifted out MSB-first

SHIFTOUT Timing



T_0 - T_1	~ 6 µs (allow pins to stabilize)
T_1 - T_2	~ 6 µs (allows receiver to capture bit)
X_1 - X_2, X_2 - X_3, ...	~ 12 µs (bit-to-bit timing)
Transmission Rate	~ 83 kBits/Sec

Here is a simple example:

```
SHIFTOUT RC.0, RC.1, MSBFIRST, 250
```

Here, the **SHIFTOUT** instruction will write to RC.0 (the *DPin*) and will generate a clock signal on RC.1 (the *CPin*). The **SHIFTOUT** instruction will generate eight clock pulses while writing each bit (of the 8-bit value 250) onto the *Dpin*. In this case, it will start with the most significant bit first as indicated by the *ShiftMode* value of MSBFIRST.

By default, **SHIFTOUT** transmits eight bits, but you can set it to shift any number of bits from 1 to 8 with the *Count* parameter. For example:

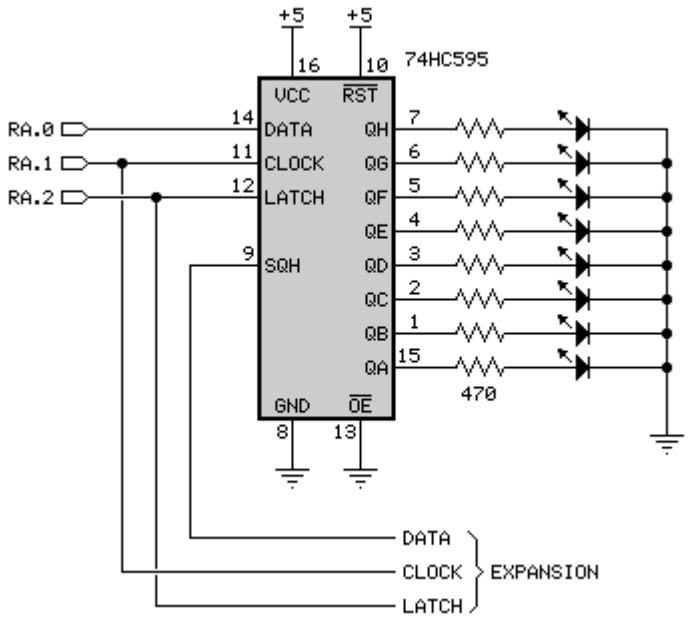
```
LOW 0  SHIFTOUT 0, 1, MSBFIRST, 250\4
```

Will output only the lowest (rightmost) four bits (%1010 in this case).

Related instruction: SHIFTIN
Related project: Thermometer

SHIFTOUT Example

Syntax




```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Transfers a counter value to eight LEDs using a 74HC595 shift register.
'
' A SpeedMult value of 10 is used to bump the SHIFTOUT clock speed to
' ~830 kBit/sec (well within 74x595 limits and 4 MHz SX clock).

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "SHIFTOUT"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

Dpin        PIN   RA.0 OUTPUT ' shift data
Cpin        PIN   RA.1 OUTPUT ' shift clock
Latch             PIN   RA.2 OUTPUT ' latch outputs

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

counter     VAR   Word              ' counter to display
tmpB1             VAR   Byte
```

```
' ============================================================================
 PROGRAM Start
' ============================================================================

' ----------------------------------------------------------------------------
' Subroutine Declarations
' ----------------------------------------------------------------------------

PUT_595     SUB   1, 2              ' allow byte or word

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:
  DO
  FOR counter = 0 TO 255                  ' loop through all values
    PUT_595 counter                       ' transfer counter to 595
    PAUSE 100                             ' wait 1/10 second
    NEXT
  LOOP

' ----------------------------------------------------------------------------
' Subroutine Code
' ----------------------------------------------------------------------------

' Use: PUT_595 value
' -- moves LSB of 'value' (can be byte or word) to 74HC595

SUB PUT_595
  tmpB1 = __PARAM1                        ' save LSB of value
  SHIFTOUT Dpin, Cpin, MSBFIRST, tmpB1, 10      ' send the bits
  PULSOUT Latch, 1                        ' transfer to outputs
ENDSUB
```

SLEEP

SLEEP

Function
Ends program execution and puts the SX into power down (sleep) mode.

Explanation
SLEEP puts the SX into power down mode halting the execution any further instructions until it is reset, either externally (via MCLR\ pin), by a watchdog timer timeout, or (if configured) a valid transition on any of the Multi-Input Wakeup (MIWU) pins (RB pins).

Related instructions: END and RESETWDT

SOUND

Example

SOUND *Pin, Note, Duration*

Function

generate square-wave tone for a specified duration.

- ***Pin*** is any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***Note*** is a byte variable/constant (0 - 127) specifying the frequency of the tone (see calculation below).
- ***Duration*** is a byte variable/constant (1 - 255) specifying the amount of time to generate the tone. The unit of time for *Duration* is 10 milliseconds.

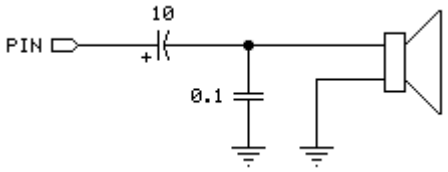
Explanation

SOUND generates one of 255 square-wave frequencies on an IO pin. The output pin should be connected as shown in the circuits below for audio use.

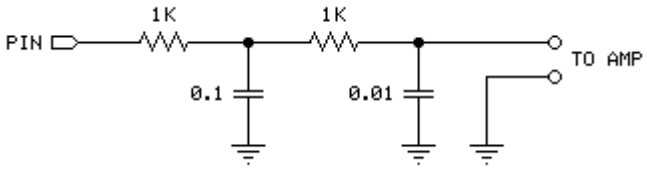
The tones produced by **SOUND** can vary in frequency from 94.3 Hz (1) to 11,062 Hz (127). If you need to determine the frequency corresponding to a given *Note* value, or need to find the note value that will give you best approximation for a given frequency, use the equations below. *Note* values above 127 are handled as *Note - 128*, in other words, a *Note* value of 138 produces the same tone as 10.

Note	INT(127 - (((1 ÷ Frequency) - 0.0000812) ÷ 0.0000805))
Frequency (Hz)	1 ÷ (0.0000812 + ((127 - Note) x 0.0000805))

When driving a Hi-Z speaker (> 40 Ω) or piezo element:



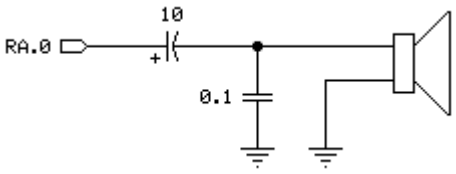
When connecting to an audio amplifier:



Related instruction: FREQOUT

SOUND Example

Syntax



40 ohm speaker or
peizo element

```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' This program generates a constant tone 25 followed by an ascending tones.
' Both the tones are played for about 100 milliseconds.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "SOUND"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

Spkr        PIN   RA.0 OUTPUT       ' piezo or 40-ohm speaker

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

tone        VAR   Byte              ' tone to play
temp1             VAR   Byte              ' for subroutine
temp2             VAR   Byte

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

PLAY        SUB   2                 ' pass note and timing

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  FOR tone = 1 TO 127
    PLAY 25, 10                     ' play tone 25
    PLAY tone, 10                         ' play ascending tone
  NEXT
```

```
  END

' -----------------------------------------------------------------------
' Subroutines Code
' -----------------------------------------------------------------------

' Use: PLAY note, duration
' -- play 'note' for 'duration' units

SUB PLAY
  temp1 = __PARAM1                          ' save note
  temp2 = __PARAM2                          ' save duration
  IF temp1 > 0 THEN
    IF temp2 > 0 THEN
      SOUND Spkr, temp1, temp2
    ENDIF
  ENDIF
ENDSUB
```

SWAP

SWAP *Variable*

Function
Exchanges the upper and lower elements of *Variable*.

- ***Variable*** is a byte (including an array element) or word variable.

Explanation
The **SWAP** instruction exchanges the upper and lower nibbles of a byte, or the upper and lower bytes of a word.

```
Main:
  RB = $3F
  SWAP RB               ' RB now holds $F3
  result = $0A55
  SWAP result                ' result now holds $550A
```

TIMER (SX48/52 only)

Example

TIMER[1 | 2] *Command {Value {, Value}}*

Function

Configures the SX48/52 Multi-Function timer T1 (TIMER1) or T2 (TIMER2).

- ***Command*** is a timer configuration command (see below).
- ***Value*** is a byte or word constant/variable as required by *Command*.

Quick Facts

I/O Pin	Function
RB.4	Timer T1 Capture Input 1
RB.5	Timer T1 Capture Input 2
RB.6	Timer T1 PWM/Compare Output
RB.7	Timer T1 External Event Clock Source
RC.0	Timer T2 Capture Input 1
RC.1	Timer T2 Capture Input 2
RC.2	Timer T2 PWM/Compare Output
RC.3	Timer T2 External Event Clock Source

TIMER Commands

TIMER[1 | 2] CLEAR

Clears the hardware timer/counter.

TIMER[1 | 2] PRESCALE *Value*

Sets the hardware timer/counter prescaler. *Value* is a constant as defined in the table below.

Value	Prescaler
0	1:1
1	1:2
2	1:4
3	1:8
4	1:16
5	1:32
6	1:64
7	1:128

TIMER[1 | 2] R1 *Value*

Sets the hardware timer/counter R1 register. *Value* is a byte or word constant/variable.

TIMER[1 | 2] R2 *Value*

Sets the hardware timer/counter R2 register. *Value* is a byte or word constant/variable.

TIMER[1 | 2] TIMER

Sets the hardware timer/counter to "Software Timer" mode.

TIMER[1 | 2] PWM *OnCycles, PeriodCycles*

Sets the hardware timer/counter to "PWM" mode. *OnCycles* is loaded into timer register R1, and *PeriodCycles* - *OnCycles* is loaded into register R2. Note that you must make the PWM pin an output. *OnCycles* and *PeriodCycles* are byte or word constants/variables.

The example below illustrates the use the of PWM option to modulate an IR LED at 38 kHz:

```
DEVICE            SX48, OSCXT1
FREQ        4_000_000

IrLed             PIN   RB.6 OUTPUT

PROGRAM Start

Start:
  TIMER1 PWM, 52, 105               ' modulate at 38 kHz, 50%
  DO
    ' additional program statements
  LOOP
  END
```

TIMER[1 | 2] CAPTURE

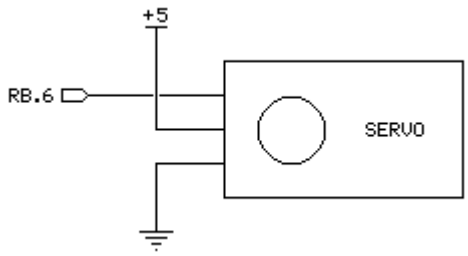
Sets the hardware timer/counter to "Capture/Compare" mode.

TIMER[1 | 2] EXTERNAL

Sets the hardware timer/counter to "External Event" mode.

TIMER Example (SX48/52 only)

Syntax




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Demonstates the SX48/52 TIMER1 in PWM mode to control a servo.
'
' Notes:
' -- 4 MHz / 32 = 125,000 Hz -> 8 uS per cycle (0.000008 s)
' -- 20 ms / 8 uS = 2500 (0.020 / 0.000008 = 2500)
' -- 1 ms / 8 uS = 125 (0.001 / 0.000008 = 125)
' -- 2 ms / 8 uS = 250 (0.002 / 0.000008 = 250)


' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX48, OSCXT1
FREQ        4_000_000
ID          "TIMER"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

Servo             PIN   RB.6 OUTPUT       ' T1 output pin

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

ServoCycle  CON   2_500                   ' 20 ms @ 4 MHz, 1:32
ServoMin    CON   125
ServoMax    CON   250

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

position    VAR   Byte              ' servo position
tmpW1             VAR   Word              ' for subroutines

' =======================================================================
 PROGRAM Start
```

```
' ============================================================================

' ----------------------------------------------------------------------------
' Subroutine Declarations
' ----------------------------------------------------------------------------

DELAY             SUB   1, 2              ' delay in milliseconds

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:
  TIMER1 PRESCALE, 5                     ' set prescaler to 1:32
  DO
    FOR position = ServoMin TO ServoMax STEP 5
      TIMER1 PWM, position, ServoCycle
      DELAY 200 ' slow sweep
    NEXT
    FOR position = ServoMax TO ServoMin STEP -5
      TIMER1 PWM, position, ServoCycle
      DELAY 20 ' faster sweep
    NEXT
  LOOP

' ----------------------------------------------------------------------------
' Subroutine Code
' ----------------------------------------------------------------------------

' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                     ' save byte value
  ELSE
    tmpW1 = __WPARAM12                   ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB
```

TOGGLE

Example

TOGGLE *BitVar*

Function
Invert the state of the specified bit.

- ***BitVar*** is a bit variable or any SX I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).

Explanation
TOGGLE inverts the state of the specified bit, changing 0 to 1 and 1 to 0. If *BitVar* is an SX I/O pin, it sets a pin to output mode and inverts the pin's state.

In some situations **TOGGLE** may appear to have no effect on a pin's state. For example, suppose RA.3 is in input mode and pulled to +5V by a 10 kΩ resistor. Then the following code executes:

```
Main:
  INPUT RA.3                  ' make RA.3 an input
  RA.3 = 0              ' set output driver to 0
  TOGGLE RA.3                 ' toggle pin state
```

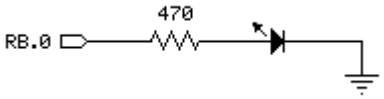
The state of RA.3 doesn't change; it's high (due to the pull-up resistor) before **TOGGLE**, and it's high (due to the pin being output high) afterward. The point is that **TOGGLE** works on associated port register bit, which may not match the pin's state when the pin is initially an input. To guarantee that the state actually changes, regardless of the initial input or output mode, do this:

```
Main:
  INPUT RA.3                  ' make RA.3 an input
  RA.3 = RA.3                 ' get state at pin
  TOGGLE RA.3                 ' toggle pin state
```

Related instructions: HIGH, LOW, and OUTPUT

TOGGLE Example

Syntax




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Simple LED blinker using TOGGLE

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "TOGGLE"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

AlarmLed    PIN   RB.0 OUTPUT       ' LED pin

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

BlinkDelay  CON   250               ' blink rate

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  DO
    PAUSE BlinkDelay                    ' delay
    TOGGLE AlarmLed                     ' toggle LED state
  LOOP                                  ' loop forever
```

TTL (SX48/52 Only)

TTL *Pin {, Enable}*

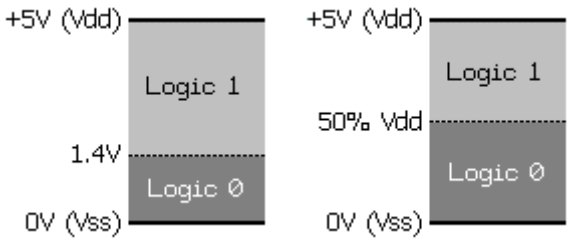
Function

Configures *Pin* for TTL input threshold (1.4 volts) on the SX48 or SX52. This command does not apply to the SX18, SX20, or SX28 (use the LVL_A, LVL_B, and LVL_C registers).

- ***Pin*** is any SX48/52 I/O pin (RA.0 .. RA.7, RB.0 .. RB.7, RC.0 .. RC.7, RD.0 .. RD.7, RE.0 .. RE.7).
- ***Enable*** is a constant, 0 or 1, that enables (1) or disables (0) the TTL input threshold. When not specified, *Enable* defaults to 1. If *Enable* is 0, the pin will be configured for CMOS input threshold.

Explanation

Every I/O pin has selectable logic level control that determines the voltage threshold for a logic level 0 or 1. The default logic level for all I/O pins is TTL but can be modified by writing to the appropriate logic-level register (LVL_A, LVL_B, LVL_C, LVL_D and LVL_E). The logic level can be configured for all pins, regardless of pin direction, but really matters only when the associated pin is set to input mode. By configuring logic levels on input pins, the SX chip can be sensitive to both TTL and CMOS logic thresholds. The figure below demonstrates the difference between TTL and CMOS logic levels.



TTL Logic Level **CMOS Logic Level**

The logic threshold for TTL is 1.4 volts; a voltage below 1.4 is considered to be a logic 0, while a voltage above is considered to be a logic 1. The logic threshold for CMOS is 50% of Vdd, a voltage below ½ Vdd is considered to be a logic 0, while a voltage above ½ Vdd is considered to be a logic 1.

```
Start:
  TTL       RE 7, 1           ' set to TTL level
  TTL       RE 6              ' set to TTL level
  TTL       RE 5, 0           ' disable TTL level, set to CMOS level
```

Related instructions: CMOS, PULLUP, and SCHMITT

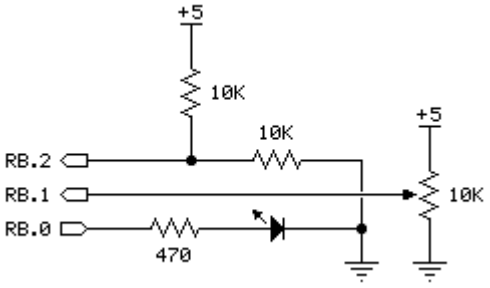
Using the Analog Comparator

I/O pins 0 through 2 in port B can be set for comparator operation. This can be configured by writing to the EN and OE bits of the Comparator register (CMP_B) and monitored by reading the RES bit. The comparator mode is disabled by default. Comparator mode can be activated for all three pins, regardless of pin direction, but really matters only when pin 1 and 2 are set to input mode (pin 0 can optionally be set to output the comparative result). By configuring Comparator mode, the SX chip can quickly determine logical differences between two signals and even indicate those differences for external circuitry.

When comparator mode is activated, the RES bit in the Comparator register indicates the result of the compare. A high bit (1) indicates the voltage on pin 2 is higher than that of pin 1, a low bit (0) indicates the voltage on pin 2 is lower than that of pin 1. If the OE bit (Output Enable) of the Comparator register is cleared, output pin 0 of port B reflects the state of the RES bit.

To configure port B I/O pins 0 though 2 for Comparator mode:

1. Set CMP_B to enable the Comparator and, optionally, the result output on pin 0.
2. Set I/O pin directions appropriately.




```
' -----------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------
'
' Demonstrates the use of the SX comparator and controlling RB.0 for
' external circuitry.

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "CMP_B"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

LED         PIN   RB.0 OUTPUT

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

result            VAR   Byte
cmpValue    VAR   result.0
```

```
' =============================================================================
 PROGRAM Start
' =============================================================================
' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  CMP_B = 0                    ' enable comparator

Main:
  CMP_B = result         ' exchange result with CMP
  GOTO Main
```

See also: COMPARE instruction and ADC8 Example.

The Elements of SX/B Style

INTRODUCTION

Like most versions of the BASIC programming language, SX/B is very forgiving and the compiler enforces no particular formatting style. As long as the source code is syntactically correct, it will usually compile and can be programmed into to the SX microcontroller without trouble.

Why, then, would one suggest a specific style for SX/B? Consider this: Millions of SX microcontrollers have been sold, SX/B makes the SX accessible to a wider audience (i.e., less-experienced programmers), and there are over 4000 members that participate in Parallax online forums. This makes it highly likely that you'll be sharing your SX/B code with someone, if not co-developing a SX-based project. Writing code in an organized, predictable manner will save you – and your potential teammates – a lot of time; in analysis, in troubleshooting, and especially when you return to a project after a long break.

The style guidelines presented here are just that: *guidelines*. They have been developed from style guidelines used by professional programmers using other high-level languages such as Visual Basic®, C/C++, and Java™. We suggest you use these guidelines as-is, or – especially if you're advanced and have been programming a while – modify them to suit your individual needs. The key is selecting a style the works well for you or your organization, and then sticking with it.

SX/B Style Guidelines

1. Do It Right The First Time

Many programmers, especially new ones, fall into the *"I'll knock it out now and fix it later."* trap. Invariably, the *"fix it later"* part never happens and sloppy code makes its way into production projects. If you don't have time to do it right, when will you find time to do it again?

Start clean and you'll be less likely to introduce errors into your code. And if errors do pop up, clean and organized formatting will make them easier to find and fix.

2. Be Organized and Consistent

Using a blank program template will help you organize your programs and establish a consistent presentation. The SX-Key IDE allows you to specify a file template for the **File | New (SX/B)** menu option.

3. Use Meaningful Names

Be verbose when naming constants, variables and program labels. The compiler will allow names up to 32 characters long. Using meaningful names will reduce the number of comments and make your programs easier to read, debug, and maintain.

4. Naming I/O Pins

Begin I/O pin names with an uppercase letter and use mixed case, using uppercase letters at the beginning of new words within the name.

```
HeaterCtrl PIN RA.0 OUTPUT
```

Since connections don't change during the program run, I/O pins are named like constants (#5) using mixed case, beginning with an uppercase letter. Resist the temptation to use direct pin names (e.g., RB.7) in the body of a program as this can lead to errors when making circuit changes.

5. Naming Constants

Begin constant names with an uppercase letter and use mixed case, using uppercase letters at the beginning of new words within the name.

```
AlarmCode CON 25
```

6. Naming Variables

Begin variable names with a lowercase letter and use mixed case, using uppercase letters at the beginning of new words within the name.

```
waterLevel VAR Byte tally VAR Word
```

7. Variable Type Declarations

SX/B supports word, byte, byte array, and bit variables. To define bit variables, the byte that holds them must be defined first.

```
sysCount    VAR   Word
alarms      VAR   Byte
overTemp    VAR   alarms
underTemp   VAR   alarms.1
clock       VAR   Byte(3)
```

8. General Program Labels

Begin program labels with an uppercase letter, used mixed case, separate words within the label with an underscore character, and begin new words with a number or uppercase letter. Labels should be preceded by at least one blank line, begin in column 1, and must be terminated with a colon.

```
Get_Tag:
  RfidEn = Active
  DO
    char = RX_RFID
  LOOP UNTIL char = $0A
  FOR idx1 = 0 TO 9
    tagBuf(idx1) = RX_RFID
  NEXT
  RfidEn = Deactivated
```

9. SX/B Keywords

All SX/B language keywords, including **CON**, **VAR**, and **SUB**should be uppercase. The SX-Key IDE does syntax highlighting, but does not change case so this is the responsibility of the programmer.

```
Main:
  DO HIGH AlarmLed
    WAIT_MS 100
    LOW AlarmLed
    WAIT_MS 100
  LOOP
```

10. Declare Subroutines and Functions

Declared subroutines were introduced in version 1.2, and as of version 1.5 SX/B now supports functions as well; functions allow a routine to return a two-byte (word) value, while subroutines are limited to returning a single byte.

Declared subroutines and functions benefits the programmer in two ways: 1) the compiler creates a jump table that allows the subroutine code to be placed anywhere in the program space and, 2) the compiler does a syntax check on the subroutine call to ensure that the proper number of parameters are being passed.

```
WAIT_MS     SUB   1, 2
GET_TEMP    FUNC  2
```

When using a declared subroutine, the use of **GOSUB** not required.

```
Main:
  DO
    HIGH AlarmLed
    WAIT_MS 100
    LOW AlarmLed
    WAIT_MS 100
    LOOP
```

11. Declared Subroutine and Function Labels

Declared subroutines and functions are, in effect, added language elements and should be treated like new keywords: all uppercase. To distinguish subroutine labels from SX/B keywords use an underscore between new words.

```
' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535


SUB DELAY  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                ' save byte value
  ELSE
    tmpW1 = __WPARAM12              ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB
```

As shown above, it is good practice to document the subroutine with usage requirements, especially when optional an parameter is available.

12. Indent Nested Code

Nesting blocks of code improves readability and helps reduce the introduction of errors. Indenting each level with two spaces is recommended to make the code readable without taking up too much space.

```
' Use: LCD_OUT [ aByte | string | label ]
' -- "aByte" is single-byte constant or variable
' -- "string" is an embedded literal string
' -- "label" is DATA statement label for stored z-String

SUB LCD_OUT
..temp1 =   PARAM1
..IF __PARAMCNT = 2 THEN
....temp2 = __PARAM2
....DO
......READ temp2 + temp1, temp3
......IF temp3 = 0 THEN EXIT
......SEROUT LcdTx, LcdBaud, temp3
......INC temp1
......temp2 = temp2 + Z
....LOOP
..ELSE
....SEROUT LcdTx, LcdBaud, temp1
..ENDIF
ENDSUB
```

Note: *The dots are used to illustrate the level of nesting and are not a part of the code.*

13. DATA / WDATA Tables

DATA / **WDATA** tables should be placed after the main code loop to prevent the program from attempting to execute these statements.

```
Main:
  IF idx = 9 THEN
    idx = 0
  ELSE
    INC idx
  ENDIF
  READ SegMap + idx, Leds
  PAUSE 1000
  GOTO Main

' -----------------------------------------------------------------------------

SegMap: ' segments maps
'        .gfedcba
  DATA  %00111111                  ' 0
  DATA  %00000110                  ' 1
  DATA  %01011011                  ' 2
  DATA  %01001111                  ' 3
  DATA  %01100110                  ' 4
  DATA  %01101101                  ' 5
  DATA  %01111101                  ' 6
  DATA  %00000111                  ' 7
  DATA  %01111111                  ' 8
  DATA  %01100111                  ' 9
```

14. Be Generous With White Space

White space (spaces and blank lines) has no effect on compiler or SX performance, so be generous with it to make listings easier to read. As suggested in #8 above, allow at least one blank line before program labels (two blanks lines before a subroutine label is recommended). Separate items in a parameter list with a space after the comma.

SX/B Error Codes

The SX/B compiler can trap several programming errors before generating the assembly output that is passed to the SASM assembler. The list below gives a description of the errors and guidance for correction.

Error 1 INVALID VARIABLE NAME
You have used a variable name that is an SX/B or SASM reserved word.

Error 2 DUPLICATE VARIABLE NAME
You have tried to allocate the same variable name more than once.

Error 3 VARIABLE EXCEEDS AVAILABLE RAM
You have declared too many variables or the array is too large.

Error 4 CONSTANT EXPECTED
The bit field value must be a constant.

Error 5 BYTE PARAMETER EXPECTED
Many commands work only with BYTE parameters.

Error 6 INVALID UNARY OPERATOR
You used a unary operator other than **–** (negate) or **~** (bitwise NOT).

Error 7 INVALID REGISTER OPERATION
Some SX registers are write-only, hence assignment is the only valid operation.

Error 8 INVALID PARAMETER
You have used a parameter that is not valid.

Error 9 SYNTAX ERROR
Command syntax not followed.

Error 10 INVALID NUMBER OF PARAMETERS
You have given too few or too many parameters for the command.

Error 11 BYTE VARIABLE EXPECTED
You have used a non-byte variable where a byte was expected.

Error 12 NOT A **FOR** CONTROL VARIABLE
You have used **NEXT** with a variable what was not used in a previous **FOR** instruction.

Error 13 BIT VARIABLE EXPECTED
You have used a non-bit variable where a bit variable is required.

Error 14 BAUDRATE IS TOO LOW
Baudrates are limited by the **FREQ** setting. Use a higher baudrate or lower clock frequency.

Error 15 BAUDRATE IS TOO HIGH
Baudrates are limited by the **FREQ** setting. Use a lower baudrate or higher clock frequency.

Error 16 UNKNOWN COMMAND
Instruction used is not supported or is misspelled, or you have misspelled a variable name.

Error 17 COMMA EXPECTED
In most cases you must separate parameters with a comma.

Error 18 EXPECTED A VALUE BETWEEN 0 AND 7
Bit operations require a value from 0 to 7.

Error 19 BIT IS NOT A HARDWARE PIN
Some commands operate only on hardware pins.

Error 20 BIT CONSTANT EXPECTED
Expected 0 or 1.

Error 21 **INTERRUPT** MUST BE USED BEFORE **PROGRAM**
Interrupt routine must be placed before the **PROGRAM** directive.

Error 22 **FOR** WITHOUT **NEXT**
You have created a **FOR** loop without a corresponding **NEXT**, or you have misspelled the **FOR** control variable name.

Error 23 **NEXT** WITHOUT **FOR**
You have used **NEXT** without a corresponding **FOR**, or you have misspelled the **NEXT** variable name.

Error 24 UNKNOWN VARIABLE NAME
You have attempted to use an undeclared variable, or misspelled a variable name.

Error 25 TOO MANY SUBS DEFINED
A maximum of 127 subroutines may be defined in any one program.

Error 26 **ELSE** OR **ENDIF** WITHOUT **IF**
You have used **ELSE** or **ENDIF** without a corresponding **IF**.

Error 27 VARIABLE NOT IN CURRENT BANK
The variable used is not accessible from the current bank.

Error 28 MUST BE A GLOBAL VARIABLE
Some commands (**BANK**) require a variable that is located in the global RAM area. Place global variables at the beginning of the declaration section.

Error 29 **LOOP** WITHOUT **DO**
You have used **LOOP** without a corresponding **DO**.

Error 30 **EXIT** NOT IN **FOR...NEXT** OR **DO...LOOP**
The **EXIT** command must be used within a **FOR...NEXT** or a **DO...LOOP** construct.

Error 31 FREQUENCY DIFFERENT FROM DEVICE SETTING
The clock speed you have specified with **FREQ** does not match the clock speed specified on the **DEVICE>** line. This only occurs when the device line specifies one of the internal clock speeds (OSC4MHZ, OSC1MHZ, OSC128KHZ, or OSC32KHZ)

Error 32 NOT ALLOWED ON THIS DEVICE
Some commands (like **TIMER1**) only apply to the SX48/52 devices. If you attempt to use these commands on the SX28 you will receive this error.

Error 33 NO **PROGRAM** DIRECTIVE USED
All SX/B programs must use the **PROGRAM Label** directive so that the SX knows where to start program execution.

SX/B Programming Template

Examples Index

The following programming template, while somewhat verbose, will help programmers new to SX/B and the SX microcontroller to keep things in order so that programs compile and assemble successfully. Parallax suggests that you use this template as is, removing unused sections only after your program is tested and working as desired.

```
' ============================================================================
'
' File...... TEMPLATE.SXB
' Purpose... SX/B Programming Template
' Author....
' E-mail....
' Started...
' Updated... 05 JUL 2006
'
' ============================================================================

' ----------------------------------------------------------------------------
' Program Description
' ----------------------------------------------------------------------------

' ----------------------------------------------------------------------------
' Device Settings
' ----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "SXB 1.50"

' ----------------------------------------------------------------------------
' IO Pins
' ----------------------------------------------------------------------------


' ----------------------------------------------------------------------------
' Constants
' ----------------------------------------------------------------------------


' ----------------------------------------------------------------------------
' Variables
' ----------------------------------------------------------------------------


' ----------------------------------------------------------------------------
 INTERRUPT
' ----------------------------------------------------------------------------

ISR_Start:
  ' ISR code here

ISR_Exit:
  RETURNINT ' {cycles}


' ============================================================================
```

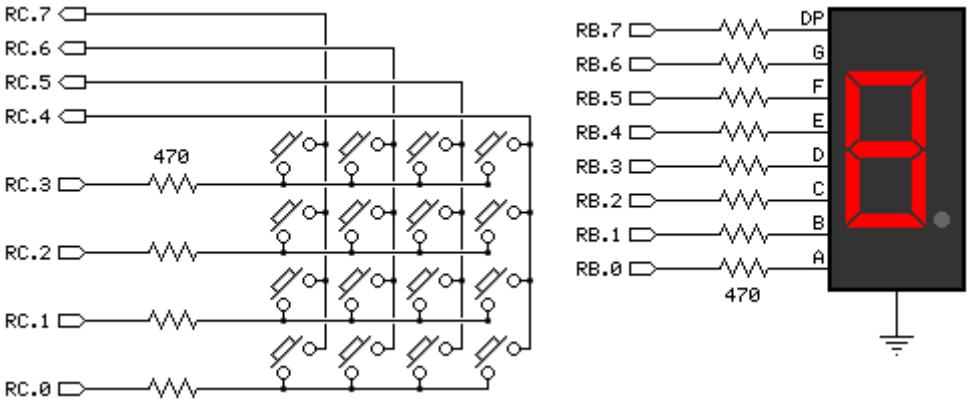
```
 PROGRAM Start
' ============================================================================


' ----------------------------------------------------------------------------
' Subroutine Declarations
' ----------------------------------------------------------------------------


' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:
  ' initialization code here

Main:
  ' main code here
  GOTO Main


' ----------------------------------------------------------------------------
' Subroutine Code
' ----------------------------------------------------------------------------


' ============================================================================
' User Data
' ============================================================================

Pgm_ID:
  DATA "SX/B 1.50 Template", 0
```

SX/B Example Projects

- Programming Template
- 8-bit ADC
- Digital Dice
- Clock / Timer
- Thermometer
- Scanning a 4x4 Matrix Keypad
- Serial LCD
- Quadrature Encoder Input
- RFID Reader Interface
- SONY IRCS Decoding

SX/B Example: 4x4 Matrix Keypad

Examples Index




```
' =====================================================================
'
' File..... KEYPAD.SXB
' Purpose... Scanning a 4x4 Matrix Keypad
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =====================================================================

' ---------------------------------------------------------------------
' Program Description
' ---------------------------------------------------------------------
'
' This program demonstrates the scanning of a 4x4 matrix keypad. If no
' key is pressed the GET_KEY routine will return a value of 16.
'
' Key values (hex):
'
' C1 C2 C3 C4
'
' R1 [ 0 ] [ 1 ] [ 2 ] [ 3 ]
'
' R2 [ 4 ] [ 5 ] [ 6 ] [ 7 ]
'
' R3 [ 8 ] [ 9 ] [ A ] [ B ]
'
' R4 [ C ] [ D ] [ E ] [ F ]
' ---------------------------------------------------------------------
' Device Settings
' ---------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "KEYPAD"

' ---------------------------------------------------------------------
' IO Pins
' ---------------------------------------------------------------------
```

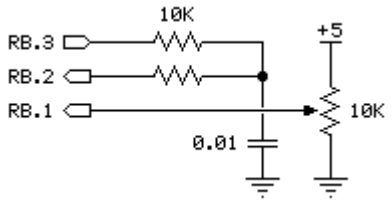
```
Keys        PIN   RC                ' keyboard scan port
TRIS_Keys   VAR   TRIS_C
PLP_Keys    VAR   PLP_C

Col1        PIN   Keys.7            ' column inputs
Col2        PIN   Keys.6
Col3        PIN   Keys.5
Col4        PIN   Keys.4

LEDs        PIN   RB OUTPUT


' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

Yes         CON   0                 ' active low input
No          CON   1

Dash        CON   %01000000         ' segment G only

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

theKey            VAR   Byte        ' from keypad, 0 - 16
row         VAR   Byte              ' keyboard scan row

tmpB1             VAR   Byte        ' subroutine work vars
tmpB2             VAR   Byte
tmpW1             VAR   Word

' =======================================================================
PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

GET_KEY     FUNC  1                 ' get key from pad
DELAY             SUB   1, 2        ' delay in milliseconds

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
 LEDs = Dash ' dash in display


Main:
  theKey = GET_KEY                  ' get a key
  IF theKey < 16 THEN               ' was a key pressed?
    READ ReMap + theKey, theKey     ' yes, remap keypad
    READ Digits + theKey, LEDs      ' output to display
    DELAY 100
  ELSE
    LEDs = Dash
  ENDIF
```

```
  GOTO Main

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' This routine works by activating each row, then scanning each column.
' If a particular row/column junction is not active (pressed), the key
' value is incremented and the scan continues. As soon as a key is found,
' the routine exits. If no key is pressed the routine will exit with a key
' value of 16.
'
' Use: aByte = GET_KEY
' -- scans keyboard and places key value into 'aByte'

FUNC GET_KEY
  tmpB1 = 0                         ' reset keyboard value
  Keys = %0000_0111                 ' activate first row
  TRIS_Keys = %1111_0000            ' refresh IO state
  PLP_Keys = %0000_1111             ' pull-up input pins

  FOR tmpB2 = 1 TO 4                ' scan four rows
    IF Col1 = Yes THEN EXIT         ' check buttons on column
    INC tmpB1                       ' update key value
    IF Col2 = Yes THEN EXIT
    INC tmpB1
    IF Col3 = Yes THEN EXIT
    INC tmpB1
    IF Col4 = Yes THEN EXIT
    INC tmpB1
    Keys = Keys >> 1                ' select next row
    Keys = Keys | %0000_1000        ' clear previous row
  NEXT
  RETURN tmpB1
ENDFUNC

' -----------------------------------------------------------------------
' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1 ' save byte value
  ELSE
    tmpW1 = __WPARAM12 ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB

' =======================================================================
' User Data
' =======================================================================

' Matrix to remap keypad values

ReMap:
  DATA 1, 2, 3, $A
  DATA 4, 5, 6, $B
  DATA 7, 8, 9, $C
  DATA $E, 0, $F, $D
```

```
' Seven-segment digit maps

Digits:
     ' .gfedcba
  DATA %00111111              ' 0
  DATA %00000110              ' 1
  DATA %01011011              ' 2
  DATA %01001111              ' 3
  DATA %01100110              ' 4
  DATA %01101101              ' 5
  DATA %01111101              ' 6
  DATA %00000111              ' 7
  DATA %01111111              ' 8
  DATA %01100111              ' 9
  DATA %01110111              ' A
  DATA %01111100              ' B
  DATA %00111001              ' C
  DATA %01011110              ' D
  DATA %01111001              ' E
  DATA %01110001              ' F
```

SX/B Example: 8-bit ADC

Examples Index

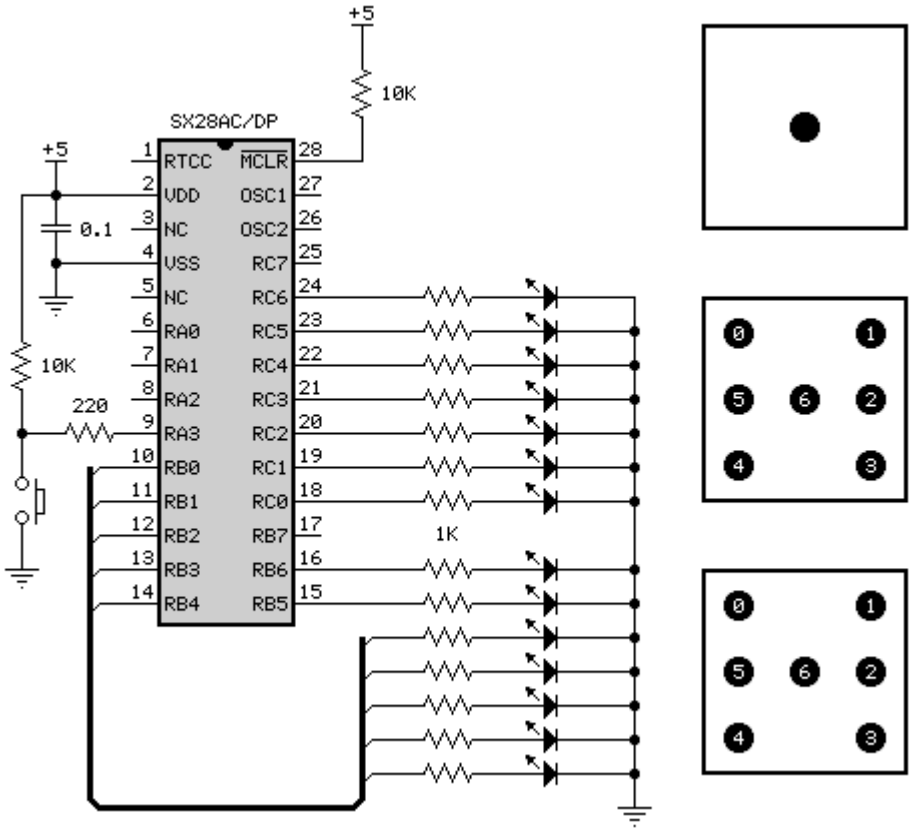



```
' =============================================================================
'
' File..... ADC8.SXB
' Purpose... Simple 8-bit ADC using the Comparator
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =============================================================================
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' This program uses an RC circuit with PWM and the SX's comparator to
' create a simple 8-bit ADC. The unknown voltage present on pin RB.1
' is found by applying a known voltage (using PWM) to pin RB.2 and
' examining the analog comparator bit.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "CMP_ADC8"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

AdcChrg     PIN   RB.3 OUTPUT ' out to RC circuit

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

adcVal            VAR   Byte        ' reading from ADC
tmpB1             VAR   Byte        ' work vars
tmpB2             VAR   Byte
tmpB3             VAR   Byte

WATCH adcVal                        ' use Debug/Poll to view

' =============================================================================
 PROGRAM Start
' =============================================================================
```

```
' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

GET_ADC     FUNC  1                 ' 8-bit ADC

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  CMP_B = 0                         ' enable comparator

Main:
  DO
    adcVal = GET_ADC                ' get new value
    BREAK ' display in Debug mode
  LOOP

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' Use: aVar = GET_ADC
' -- returns 8-bit value of voltage on RB.1

FUNC GET_ADC
  tmpB1 = 0                         ' reset result
  tmpB2 = 128                       ' bias to middle
  DO
    tmpB1 = tmpB1 + tmpB2           ' create test value
    PWM AdcChrg, tmpB1, 1           ' charge RC
    CMP_B = tmpB3                   ' compare inputs
    IF tmpB3.0 = 1 THEN             ' if unknown lower
      tmpB1 = tmpB1 - tmpB2         ' reduce test value
    ENDIF
    tmpB2 = tmpB2 >> 1              ' divide bias
  LOOP UNTIL tmpB2 = 0
  RETURN tmpB1                      ' return ADC value
ENDFUNC
```

SX/B Example: Digital Dice

Examples Index




```
' ================================================================================
'
' File...... DICE.SXB
' Purpose... A Pair of Digital Dice
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' ================================================================================

' --------------------------------------------------------------------------------
' Program Description
' --------------------------------------------------------------------------------
'
' Simple digital dice program. Uses outputs from RB and RC to dice pattern
' on seven LEDs (for each port) as shown below:
'
' (0) (1)
' (5) (6) (2)
' (4) (3)
'
' A button input on RA.0 is used to "roll" the dice. When rolling is
' stopped, display will stay solid for at least one second, then the
' program will wait for the Roll button to be pressed again.
```

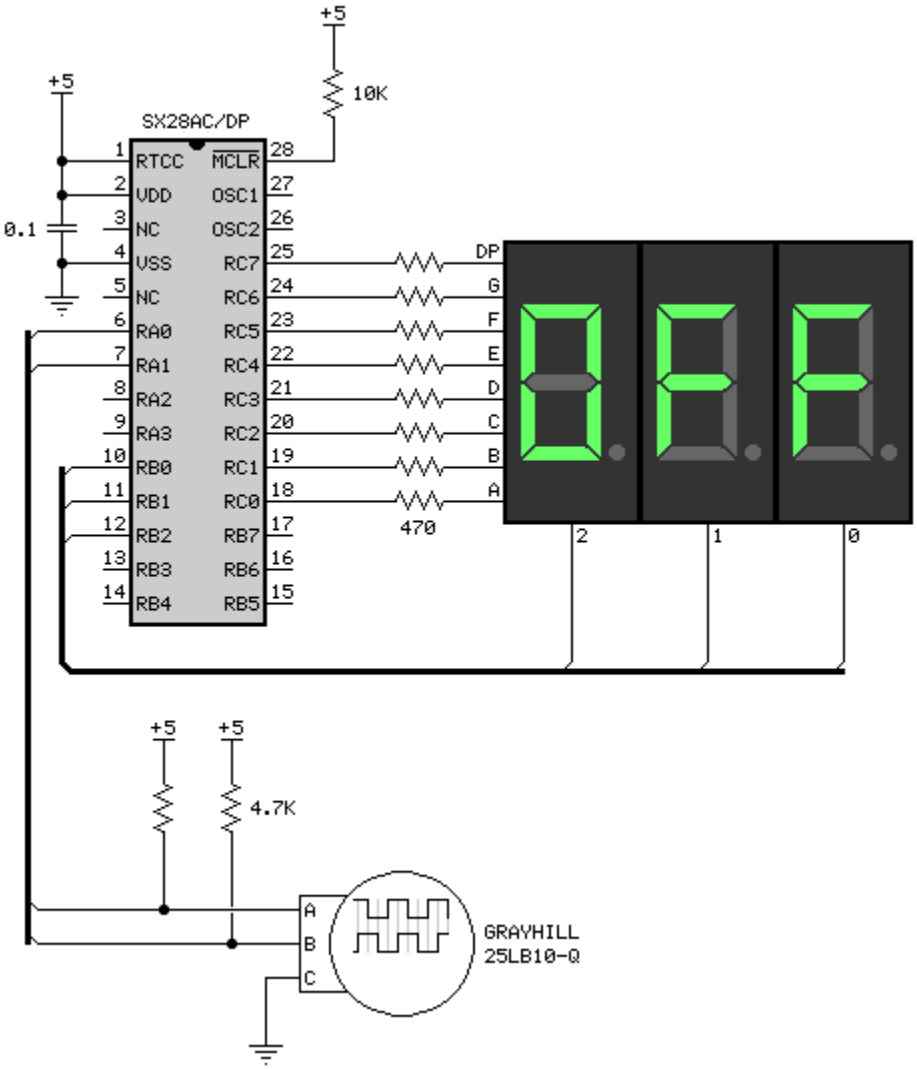
```
' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------


DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "DICE"


' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------


Roll        PIN   RA.3 INPUT  ' roll button
Die0        PIN   RB   OUTPUT ' LEDs out for die #1
Die1        PIN   RC   OUTPUT ' LEDs out for die #2


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------


No          CON   1           ' button not pressed
Yes         CON   0


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------


d1Val             VAR   Byte        ' value of die #1
d2Val             VAR   Byte        ' value of die #2
pattern     VAR   Byte        ' dice pattern

tmpB1             VAR   Byte        ' work variables
tmpB2             VAR   Byte
tmpB3             VAR   Byte
tmpW1             VAR   Word


' =============================================================================
 PROGRAM Start
' =============================================================================


' -----------------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------------


GET_DIE           FUNC 1,1          ' pass seed and index addr
DELAYS            SUB 1,2           ' delay in milliseconds


' -----------------------------------------------------------------------------
' Program Code
```

```
' -----------------------------------------------------------------------


Start:
  TRIS_B = %10000000                ' make LED ports outputs
  TRIS_C = %10000000
  PLP_A = %1000                  ' pull-up unused pins
  PLP_B = %01111111
  PLP_C = %01111111
  d1Val = $12                       ' initialize seeds
  d2Val = $34


Main:
  DO
    Die0 = GET_DIE @d1Val           ' randomize die values
    Die1 = GET_DIE @d2Val
    DELAY 75                        ' delay between rolls
  LOOP WHILE Roll = No              ' wait for button
  DELAY 1000                        ' show dice (1 sec min)


Wait_For_Press:
  DO
    DELAY 10
  LOOP UNTIL Roll = Yes

Wait_For_Release:
  DO WHILE Roll = Yes
    DELAY 10
  LOOP
  DELAY 100
  GOTO Main


' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------


' Use: pattern = GETDIE @seed
' -- randomizes 'seed' (must pass address as 'seed' is updated)
' -- returns die display in 'pattern'

FUNC GET_DIE
  tmpB1 = __PARAM1                  ' get seed address
  tmpB2 = __RAM(tmpB1)              ' get seed value
  RANDOM tmpB2                      ' randomize seed
  __RAM(tmpB1) = tmpB2              ' update seed
  tmpB2 = tmpB2 / 43                ' make = 0 to 5
  READ Pips + tmpB2, tmpB1 ' get LED pattern
  RETURN tmpB1 ' return pattern
ENDFUNC


' -----------------------------------------------------------------------


' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535
```

```
SUB DELAY
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                ' save byte value
  ELSE
    tmpW1 = __WPARAM12              ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB



' ===========================================================================
' User Data
' ===========================================================================


Pips:
  DATA %01000000              ' 1
  DATA %00010010              ' 2
  DATA %01010010              ' 3
  DATA %00011011              ' 4
  DATA %01011011              ' 5
  DATA %00111111              ' 6
```

SX/B Example: Quadrature Encoder Input

Examples Index




```
' =====================================================================
'
' File..... ENCODER.SXB
' Purpose... Grayhill 25LB10-Q Encoder Demo
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =====================================================================

' ---------------------------------------------------------------------
' Program Description
' ---------------------------------------------------------------------
'
' Reads a Grayhill 25LB10-Q quadrature encode and displays a value on
' three 7-segment LEDs. When the value is zero, the display reads "OFF."
```

```
' Reading the encoder and multiplexing the displays is handled in the
' "background" with an ISR.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "ENCODER"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

EncPort     PIN   RA                      ' encoder port
TRIS_Enc    VAR   TRIS_A
Cathodes    PIN   RB                      ' LED cathodes
TRIS_Cath   VAR   TRIS_B
Anodes            PIN   RC                ' LED anodes
TRIS_Ano    VAR   TRIS_C

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Blank             CON   %00000000         ' all segs off
Ltr_O             CON   %00111111         ' pattern for "O"
Ltr_F             CON   %01110001         ' pattern for "F"

Yes         CON   1
No          CON   0

MaxVal            CON   100

' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

display     VAR   Byte(3)                ' multiplexed segments
digPntr     VAR   Byte                   ' digit pointer
digLimit    VAR   Byte                   ' 0 - 2

flags             VAR   Byte
newVal            VAR   flags.0

encCheck    VAR   Byte
encOld            VAR   Byte             ' previous encoder bits
encNew            VAR   Byte             ' new encoder bits
encValue    VAR   Byte                   ' encoder value


tmpB1             VAR   Byte
tmpB2             VAR   Byte
tmpB3             VAR   Byte


' -----------------------------------------------------------------------------
 INTERRUPT
' -----------------------------------------------------------------------------
```

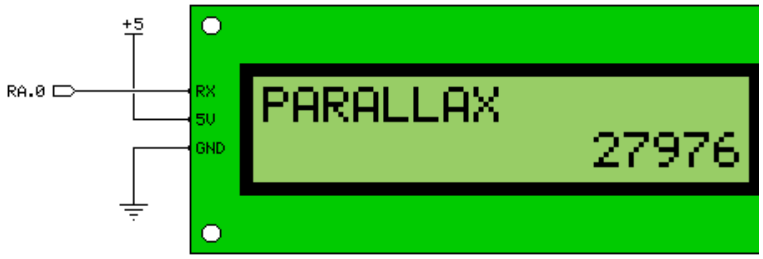
```
' Runs every 64 uS @ 4 MHz (no prescaler)

ISR_Start:
  encNew = EncPort & %00000011            ' get econder bits
  tmpB3 = encOld XOR encNew               ' test for change
  IF tmpB3 > 0 THEN                       ' change?
    encOld = encOld << 1                  ' adjust old bits
    encOld = encOld XOR encNew            ' test direction
    IF encOld.1 = 1 THEN
      IF encValue < MaxVal THEN           ' if max, no change
        INC encValue                      ' increase value
        newVal = 1                        ' alert "foreground"
      ENDIF
    ELSE
      IF encValue > 0 THEN                ' if 0, no change
        DEC encValue                      ' decrease value
        newVal = 1                        ' alert "foreground"
      ENDIF
    ENDIF
    encOld = encNew                       ' save last input
  ENDIF

Update_Display:
  INC digPntr                             ' point to next digit
  IF digPntr = digLimit THEN              ' update digit
    digPntr = 0                           ' wrap if needed
  ENDIF
  Anodes = Blank                          ' blank display
  READ DigCtrl + digPntr, Cathodes        ' select display element
  Anodes = display(digPntr)               ' output new digit segs

ISR_Exit:
  RETURNINT

' ============================================================================
 PROGRAM Start
' ============================================================================

' ----------------------------------------------------------------------------
' Subroutine Declarations
' ----------------------------------------------------------------------------

UPDATE_ABUF SUB 0 ' update anode buffer

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:
  Anodes = Blank                          ' clear LEDs
  TRIS_Ano = %00000000                    ' make anode ctrl outputs
  Cathodes = %111                         ' all digits off
  TRIS_Cath = %11111000                   ' make cath ctrl outputs
  encNew = EncPort & %00000011            ' read encoder pos
  encOld = encNew                         ' copy
  encValue = 0                            ' clear value
  UPDATE_ABUF                             ' update display char buf
  OPTION = $88                            ' interrupt, no prescaler
```

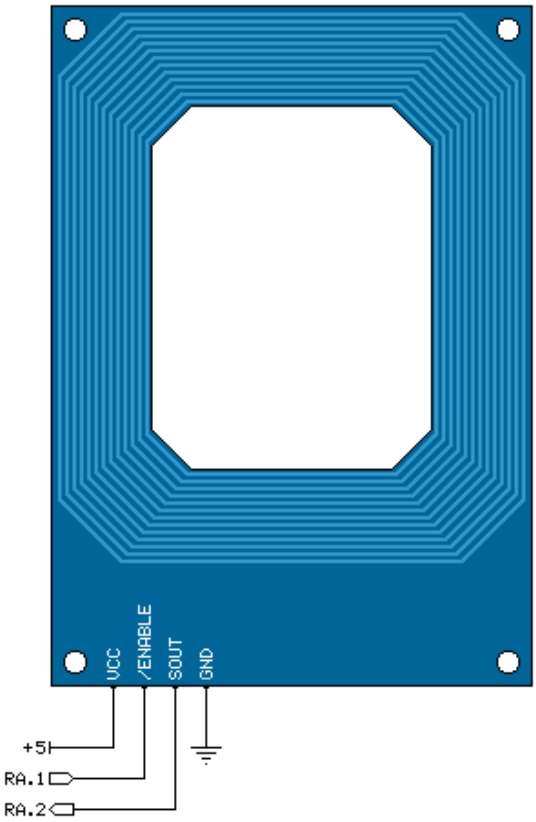
```
Main:
  DO
    IF newVal = Yes THEN                  ' check flag
      UPDATE_ABUF                         ' refresh display buffer
    ENDIF
  LOOP

' -----------------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------------

SUB UPDATE_ABUF
  IF encValue = 0 THEN
    PUT display(0), Ltr_F, Ltr_F, Ltr_O   ' if 0, show "OFF"
    digLimit = 3
  ELSE
    digLimit = 1                          ' at least one digit
    IF encValue > 9 THEN                  ' into 10s?
      INC digLimit                        ' -- yes, show another digit
    ENDIF
    IF encValue > 99 THEN                 ' into 100s?
      INC digLimit                        ' -- yes, show another digit
    ENDIF
    tmpB1 = encValue                      ' convert value to segments
    tmpB2 = tmpB1 / 100
    tmpB1 = __REMAINDER
    READ SegMaps + tmpB2, display(2)
    tmpB2 = tmpB1 / 10
    tmpB1 = __REMAINDER
    READ SegMaps + tmpB2, display(1)
    READ SegMaps + tmpB1, display(0)
  ENDIF
  newVal = No                             ' clear flag
ENDSUB

' =============================================================================
' User Data
' =============================================================================

SegMaps: ' segments maps
'      .gfedcba
  DATA %00111111 ' 0
  DATA %00000110 ' 1
  DATA %01011011 ' 2
  DATA %01001111 ' 3
  DATA %01100110 ' 4
  DATA %01101101 ' 5
  DATA %01111101 ' 6
  DATA %00000111 ' 7
  DATA %01111111 ' 8
  DATA %01100111 ' 9
  DATA %01110111 ' A
  DATA %01111100 ' b
  DATA %00111001 ' C
  DATA %01011110 ' d
  DATA %01111001 ' E
  DATA %01110001 ' F

DigCtrl:
  DATA %11111110                    ' column 0 on
```

```
DATA %11111101
DATA %11111011
```

SX/B Example: RFID Reader Interface

Examples Index






```
' =============================================================================
'
' File..... RFID_SECURITY.SXB.SXB
' Purpose... Parallax RFID Reader Demo
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =============================================================================
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Simple security application using the Parallax RFID reader and the
' Parallax Serial LCD. As designed, the application will support 16
' RFID tags.
'
' Components:
'
'  Parallax RFID Reader... #28140
'  Parallax Serial LCD.... #27976 or #27977

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------
```

```
DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "RFID"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

LcdTx             PIN   RA.0 OUTPUT       ' LCD serial connection
RfidEn            PIN   RA.1 OUTPUT       ' RFID enable control
RfidRx            PIN   RA.2 INPUT        ' RFID serial input
Lock        PIN   RA.3 OUTPUT       ' lock control

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

TagMax            CON   2                 ' three tags, (0 - 2)

LcdBaud     CON   "T19200"          ' or T2400, or T9600
RfidBaud    CON   "T2400"

LcdBkSpc    CON   $08               ' move cursor left
LcdRt             CON   $09               ' move cursor right
LcdLF             CON   $0A               ' move cursor down 1 line
LcdCls            CON   $0C               ' clear LCD (need 5 ms delay)
LcdCR             CON   $0D               ' move pos 0 of next line
LcdBLon     CON   $11               ' backlight on
LcdBLoff    CON   $12               ' backlight off
LcdOff            CON   $15               ' LCD off
LcdOn1            CON   $16               ' LCD on; no crsr, no blink
LcdOn2            CON   $17               ' LCD on; no crsr, blink on
LcdOn3            CON   $18               ' LCD on; crsr on, no blink
LcdOn4            CON   $19               ' LCD on; crsr on, blink on
LcdLine1    CON   $80               ' move to line 1, column 0
LcdLine2    CON   $94               ' move to line 2, column 0

Active            CON   0                 ' for RFID reader
Deactivated       CON   1

Open        CON   1                 ' for lock
Closed            CON   0

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

idx1        VAR   Byte              ' loop control
idx2        VAR   Byte
char        VAR   Byte

tagBuf            VAR   Byte(10)          ' tag bytes from reader
tagNum            VAR   Byte              ' tag number
offset            VAR   Byte

tmpB1             VAR   Byte              ' subroutine work vars
tmpB2             VAR   Byte
tmpB3             VAR   Byte
tmpW1             VAR   Word
```

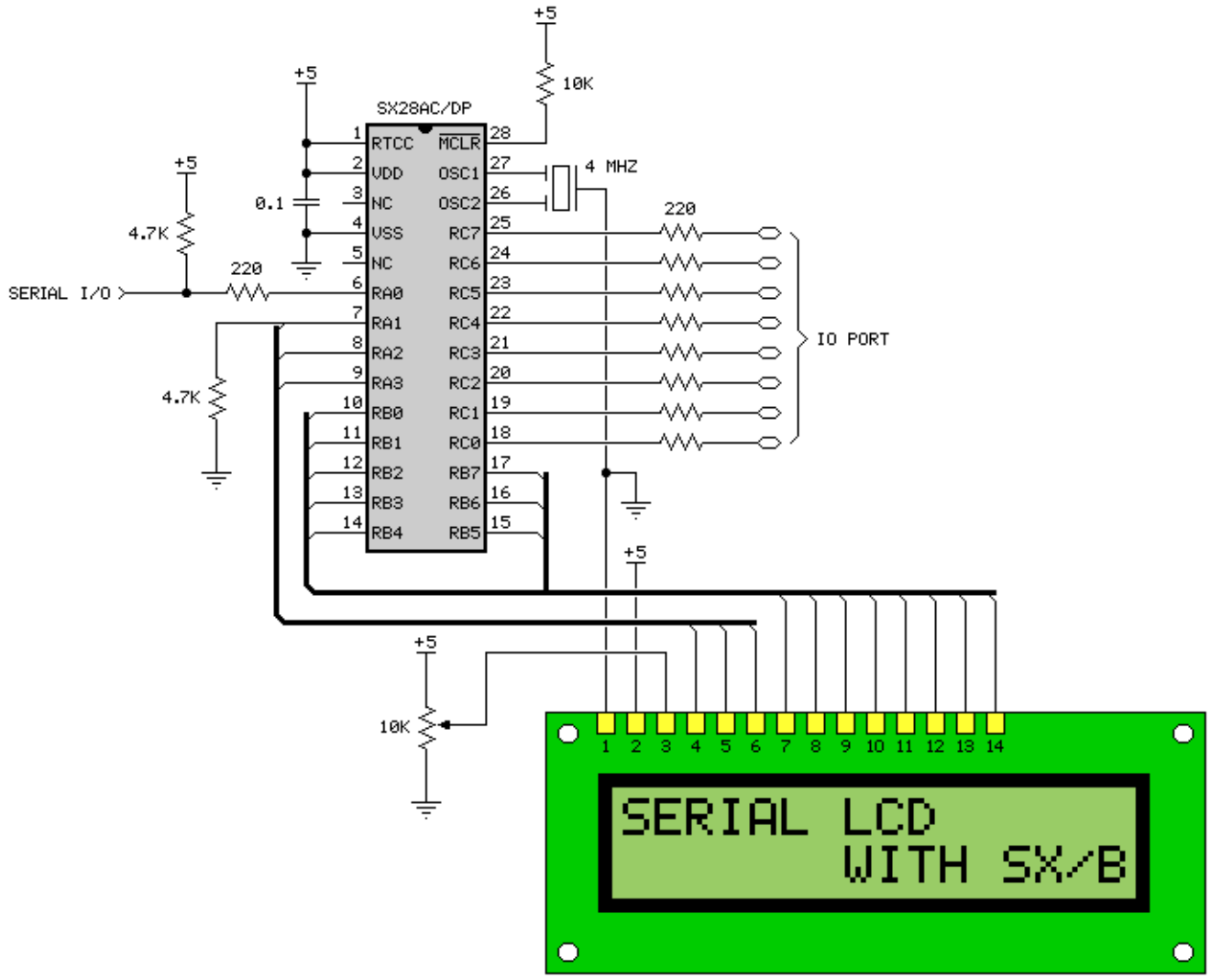
```
' =============================================================================
 PROGRAM Start
' =============================================================================

' -----------------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------------

DELAY            SUB   1, 2             ' delay in milliseconds
LCD_OUT     SUB   1, 2             ' byte or string to LCD
CLEAR_LCD   SUB   0                ' clear LCD, BL is on
RX_RFID     FUNC  1                ' get char from RFID

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  PLP_B = %00000000                     ' pull up unused pins
  PLP_C = %00000000
  RA = %0011                            ' disable reader, lock it up
  TRIS_A = %0100
  DELAY 100                             ' let LCD initialize

Main:
  CLEAR_LCD
  LCD_OUT "Present ID."
  LCD_OUT LcdLine2
  LCD_OUT LcdOn2                    ' flash block cursor

Get_Tag:
  RfidEn = Active
  DO
    char = RX_RFID                      ' get a character
  LOOP UNTIL char = $0A                 ' wait for header
  FOR idx1 = 0 TO 9                     ' get RFID bytes
    tagBuf(idx1) = RX_RFID
  NEXT
  RfidEn = Deactivated

Search_Tags:
  FOR tagNum = 0 TO TagMax              ' loop through known tags
    offset = tagNum * 10                ' point to tag string
    FOR idx1 = 0 TO 9                   ' loop through characters
      READ Tags + offset, char                ' read tag character
      INC offset ' point to next
      IF char <> tagBuf(idx1) THEN Next_Tag     ' if bad, skip rest
    NEXT
    GOTO Found_Tag                      ' if all valid, tag found

Next_Tag:
  NEXT

No_Tag:
  CLEAR_LCD
  LCD_OUT "Unauthorized"
  GOTO Loop_Pad

Found_Tag:
```

```
  CLEAR_LCD
  LCD_OUT "Authorized"

Show_Name:
  LCD_OUT LcdLine2
  offset = tagNum << 4                    ' point to start of name
  FOR idx1 = 0 TO 15
    READ TagNames + offset, char          ' get name character
    INC offset                            ' point to next
    LCD_OUT char                    ' send char to LCD
  NEXT
  Lock = Open

Loop_Pad:
  DELAY 3000                              ' pause 3 seconds
  Lock = Closed
  GOTO Main

' -----------------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------------

' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                      ' save byte value
  ELSE
    tmpW1 = __WPARAM12                    ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB

' -----------------------------------------------------------------------------

' Use: LCD_OUT [ aByte | string | label ]
' -- "aByte" is single-byte constant or variable
' -- "string" is an embedded literal string
' -- "label" is DATA statement label for stored z-String

SUB LCD_OUT
  tmpB1 = __PARAM1                        ' byte or string offset
  IF __PARAMCNT = 2 THEN                  ' string specified?
    tmpB2 = __PARAM2                      ' yes, save base
    DO
      READ tmpB2 + tmpB1, tmpB3           ' read a character
      IF tmpB3 = 0 THEN EXIT ' if 0, string complete
      SEROUT LcdTx, LcdBaud, tmpB3        ' send the byte
      INC tmpB1 ' point to next character
      tmpB2 = tmpB2 + Z                   ' update base on overflow
    LOOP
  ELSE
    SEROUT LcdTx, LcdBaud, tmpB1          ' send the byte
  ENDIF
ENDSUB

' -----------------------------------------------------------------------------

' Use: CLEAR_LCD
```

```
' -- clears the LCD and activates the backlight
' -- removes cursor and blinking block

SUB CLEAR_LCD
  LCD_OUT LcdBLon                         ' backlight on
  LCD_OUT LcdOn1                    ' no cursor or blink
  LCD_OUT LcdCls                    ' clear the LCD
  DELAY 5
ENDSUB

' -----------------------------------------------------------------------

' Use: aByte = RX_RFID
' -- receives one serial byte from RFID reader

FUNC RX_RFID
  SERIN RfidRx, RfidBaud, tmpB1
  RETURN tmpB1
ENDFUNC

' =======================================================================
' User Data
' =======================================================================

Tags:
  DATA "0415148AF1"                       ' valid tags
  DATA "041514A4E0"
  DATA "04151496D8"

' Keep tag names 16-chars in length
' -- name order must match tag order

TagNames:
  DATA "Luke Skyjogger "
  DATA "Princess Leggo "
  DATA "Derth Wader "
```

SX/B Example: Serial LCD

Examples Index




```
' =============================================================================
'
' File..... SERIAL_LCD.SXB
' Purpose... Serial LCD Controller with 8 I/O bits
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =============================================================================
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Implements a simple serial LCD controller using the SX28.
'
' This controller is designed to be command-compatible with SEETRON 2x16
```

```
' LCD displays, but is fixed at 9600 baud and uses open-true serial mode.
' It also offers extended functions with RC of the SX28
'
' By sending the instruction prefix ($FE) twice, this LCD controller
' enters "extended" function mode. The following extended functions are
' currently available:
'
' $F0, ddr : Set IO port DDR (1 = input, 0 = output; default is %11111111)
' $F1 : Read IO port
' $F2, bits : Write bits to IO port
' $FF : Read LCD RAM (from current cursor position)

' -----------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "SLCD"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

Sio         PIN   RA.0 INPUT        ' pull-up via 4.7K
LcdE        PIN   RA.1              ' pull-down via 4.7K
LcdRW             PIN   RA.2
LcdRS             PIN   RA.3
TRIS_Ctrl   VAR   TRIS_A

LcdBus            PIN   RB                ' RB.0 --> DB0, ...
TRIS_Lcd    VAR   TRIS_B

IoPort            PIN   RC
TRIS_IO     VAR   TRIS_C

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

Baud        CON   "OT9600"          ' open for single wire
Cmd         CON   $FE               ' LCD command prefix

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

serByte     VAR   Byte              ' serial IO byte

tmpB1             VAR   Byte              ' work vars
tmpB2             VAR   Byte
tmpW1             VAR   Word

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------
```

```
DELAY             SUB   1, 2              ' delay in milliseconds
DELAY_US    SUB   1, 2              ' delay in microseconds
BLIP        SUB   0                 ' move bus to/from LCD
LCDINIT     SUB   0                 ' initialize LCD
LCDCMD            SUB   1                 ' command byte --> LCD
LCDOUT            SUB   1                 ' byteVar --> LCD
LCD_PUT           SUB   1                 ' byteVar --> LCD
LCDIN             FUNC  1                 ' byteVar <-- LCD
RX_BYTE     FUNC  1                 ' rx from serial I/O
TX_BYTE     SUB   1                 ' tx to serial I/O

' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  TRIS_Ctrl = %0001                     ' RA.0 = input (serial)
  LCDINIT

Main:
  serByte = RX_BYTE                     ' wait for byte
  IF serByte = Cmd THEN Do_Command      ' command?

Do_Write:
  LCDOUT serByte                    ' no, write char byte
  GOTO Main

Do_Command:
  serByte = RX_BYTE                     ' get command byte
  IF serByte = Cmd THEN Extended_Cmd           ' extended command?
  LCDCMD serByte                    ' no, write cmd byte
  GOTO Main

Extended_Cmd:
  serByte = RX_BYTE                     ' get extended command

Set_IO:
  IF serByte <> $F0 THEN Read_IO        ' check command
  serByte = RX_BYTE                     ' get port configuration
  TRIS_IO = serByte                     ' setup IO port
  GOTO Main

Read_IO:
  IF serByte <> $F1 THEN Write_IO
  DELAY 2                           ' delay for host setup
  TX_BYTE IoPort                    ' send port bits to host
  GOTO Main

Write_IO:
 IF serByte <> $F2 THEN Read_Byte
  IoPort = RX_BYTE                      ' put bits on the port
  GOTO Main
Read_Byte:
  IF serByte <> $FF THEN Cmd_X
  serByte = LCDIN                       ' read LCD (cursor pos)
  DELAY 2                           ' delay for host setup
  TX_BYTE serByte                       ' send byte to to host
  GOTO Main
```

```
Cmd_X:                                  ' for future expansion
  '
  GOTO Main

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' Use: DELAY ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                    ' save byte value
  ELSE
    tmpW1 = __WPARAM12                  ' save word value
  ENDIF
  PAUSE tmpW1
ENDSUB

' -----------------------------------------------------------------------

' Use: DELAY us
' -- 'us' is delay in microseconds, 1 - 65535

SUB DELAY_US
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                    ' save byte value
  ELSE
    tmpW1 = __WPARAM12                  ' save word value
  ENDIF
  PAUSEUS tmpW1
ENDSUB

' -----------------------------------------------------------------------

' Use: LCDCMD cmdByte
' -- send 'cmdByte' to LCD with RS line low

SUB LCDCMD                          ' write command byte
  LcdRS = 0
  Lcd_Put __PARAM1
ENDSUB

' -----------------------------------------------------------------------

' Use: LCDOUT char
' -- send 'char' to LCD with RS line high

SUB LCDOUT ' write character byte
  LcdRS = 1
  Lcd_Put __PARAM1
ENDSUB __PARAM1
' -----------------------------------------------------------------------

SUB Lcd_Put                             ' byte --> LCD bus
  LcdBus = __PARAM1
  BLIP
ENDSUB
```

```
' -----------------------------------------------------------------------

SUB BLIP                            ' move bus into LCD
  LcdE = 1
  DELAY_US 2
  LcdE = 0
  DELAY_US 40                             ' instruction delay
ENDSUB

' -----------------------------------------------------------------------

' Use: char = LCDIN
' -- puts value at LCD cursor position in 'char'

FUNC LCDIN
  TRIS_Lcd = %11111111                    ' make LCD bus inputs
  LcdRS = 1                               ' character mode
  LcdRW = 1                               ' read mode
  LcdE = 1                            ' move byte to bus
  tmpB1 = LcdBus                      ' grab the byte
  LcdE = 0                            ' complete the read
  LcdRW = 0                               ' return to write mode
  TRIS_Lcd = %00000000                    ' make LCD bus outputs
  RETURN tmpB1                            ' return byte to caller
ENDFUNC

' -----------------------------------------------------------------------

' Use: LCDINIT
' -- initialize LCD for 8-bit, 2-line interface

SUB LCDINIT:
  DELAY 15                            ' power-up delay, 15 ms
  TRIS_Lcd = %00000000                    ' all outputs
  LcdBus = %00110000                      ' 8-bit interface
  BLIP
  DELAY 5                             ' delay 4.5 ms (min)
  BLIP
  DELAY_US 100
  BLIP
  LCDCMD %00111000                        ' multi-line, 5x7 font
  LCDCMD %00001100                        ' display on, no cursor
  LCDCMD %00000110                        ' auto-increment cursor
  LCDCMD %00000001                        ' clear and home LCD
ENDSUB

' -----------------------------------------------------------------------

' Use: char = RX_BYTE
' -- reads byte from serial input and places in 'char'

FUNC RX_BYTE
  SERIN Sio, Baud, tmpB1                  ' receive a byte
  RETURN tmpB1                            ' return to caller
ENDFUNC

' -----------------------------------------------------------------------

' Use: TX_BYTE char
' -- transmits 'char' over serial connnection
```

```
SUB TX_BYTE
  tmpB1 = __PARAM1                          ' save byte to send
  SEROUT Sio, Baud, tmpB1                   ' send the byte
ENDSUB
```

The following program demonstrates the features of this serial LCD using a BASIC Stamp 2 microcontroller.

```
' ===================================================================
'
' File...... SERIAL_LCD_TEST.BS2
' Purpose... Test the SX/B Serial LCD controller
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' {$STAMP BS2}
' {$PBASIC 2.5}
'
' ===================================================================


' -----[ Program Description ]---------------------------------------
'
' This program demostrates the SX/B serial LCD controller code. The SX/B
' LCD controller is code-compatible with SEETRON LCD displays, but adds
' extended features.
'
' To access extended features, the intruction commnad (254) is sent twice,
' then followed with the extended command and possible data byte.
'
' Extended commands:
'
' <$FE><$FE><$F0>       - Set SX/B IO port DDR (1 = input, 0 = output)
' <$FE><$FE><$F1>       - Read SX/B IO port
' <$FE><$FE><$F2><bits> - Write bits to SX/B IO port
' <$FE><$FE><$FF>       - Read LCD RAM (current cursor position)


' -----[ Revision History ]------------------------------------------


' -----[ I/O Definitions ]-------------------------------------------

Sio         PIN   15


' -----[ Constants ]-------------------------------------------------

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    T1200   CON   813
    T2400   CON   396
    T4800   CON   188
    T9600   CON   84
    T19K2   CON   32
    TMidi   CON   12
    T38K4   CON   6
```

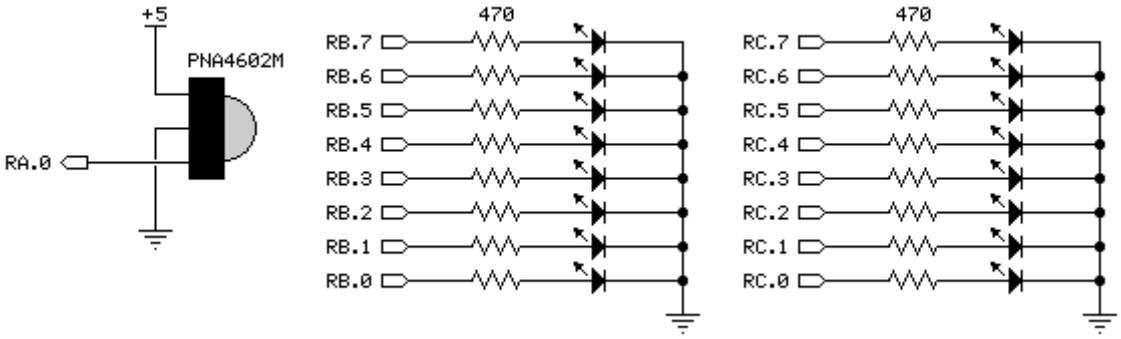
```
  #CASE BS2SX, BS2P
    T1200   CON   2063
    T2400   CON   1021
    T4800   CON   500
    T9600   CON   240
    T19K2   CON   110
    TMidi   CON   60
    T38K4   CON   45
  #CASE BS2PX
    T1200   CON   3313
    T2400   CON   1646
    T4800   CON   813
    T9600   CON   396
    T19K2   CON   188
    TMidi   CON   108
    T38K4   CON   84
  #ENDSELECT

Inverted    CON   $4000
Open        CON   $8000
Baud        CON   Open | T9600

LcdCls      CON   $01              ' clear the LCD
LcdHome     CON   $02              ' move cursor home
LcdCrsrL    CON   $10              ' move cursor left
LcdCrsrR    CON   $14              ' move cursor right
LcdDispL    CON   $18              ' shift chars left
LcdDispR    CON   $1C              ' shift chars right
LcdDDRam    CON   $80              ' Display Data RAM control
LcdCGRam    CON   $40              ' Character Generator RAM
LcdLine1    CON   $80              ' DDRAM address of line 1
LcdLine2    CON   $C0              ' DDRAM address of line 2

LcdSetIO    CON   $F0              ' set LCD IO port DDR
LcdRdIO     CON   $F1              ' read from LCD IO port
LcdWrIO     CON   $F2              ' write to LCD IO port
LcdI        CON   $FE              ' instruction
LcdRdRAM    CON   $FF              ' read from cursor pos


' -----[ Variables ]-------------------------------------------------------

eePntr      VAR   Word
char        VAR   Byte
col         VAR   Byte
idx         VAR   Nib
ioPort      VAR   Byte


' -----[ EEPROM Data ]-----------------------------------------------------

CC0 DATA $0E,$1F,$1C,$18,$1C,$1F,$0E,$00
CC1 DATA $0E,$1F,$1F,$18,$1F,$1F,$0E,$00
CC2 DATA $0E,$1F,$1F,$1F,$1F,$1F,$0E,$00


' -----[ Initialization ]--------------------------------------------------

Reset:
 PAUSE 500                      ' let LCD get ready
```

```
  SEROUT Sio, Baud, [LcdI, LcdCls]
  PAUSE 2

DL_Chars:                           ' download custom chars
  SEROUT Sio, Baud, [LcdI, LcdCGRam]
  FOR eePntr = CC0 TO (CC2 + 7)
    READ eePntr, char
    SEROUT Sio, Baud, [char]
  NEXT
  SEROUT Sio, Baud, [LcdI, LcdDDRam]


' -----[ Program Code ]----------------------------------------------------

Main:
  DEBUG CLS, "SX/B Serial LCD Demo"
  SEROUT Sio, Baud, [LcdI, LcdCls]
  PAUSE 500

Write_To_LCD:
  DEBUG CRSRXY, 0, 2, "Writing to LCD"
  SEROUT Sio, Baud, 10, ["SERIAL LCD"]
  SEROUT Sio, Baud, 10, [LcdI, LcdLine2 + 7, "WITH SX/B"]
  PAUSE 1000

Read_From_LCD:
  DEBUG CRSRXY, 0, 4, "Reading LCD L1: ["
  SEROUT Sio, Baud, [LcdI, LcdHome]
  PAUSE 2
  FOR col = 0 TO 15
    SEROUT Sio, Baud, [LcdI, LcdI, LcdRdRAM]
    SERIN Sio, Baud, [char]
    DEBUG char
  NEXT
  DEBUG "]",
  CRSRXY, 0, 5, "Reading LCD L2: ["
  SEROUT Sio, Baud, [LcdI, LcdLine2]
  PAUSE 2
  FOR col = 0 TO 15
    SEROUT Sio, Baud, [LcdI, LcdI, LcdRdRAM]
    SERIN Sio, Baud, [char]
    DEBUG char
  NEXT
  DEBUG "]"
  PAUSE 1000

Animation:
  DEBUG CRSRXY, 0, 7, "Custom character demo"
  FOR col = LcdLine2 TO (LcdLine2 + 15)
    FOR idx = 0 TO 5
      SEROUT Sio, Baud, [LcdI, col]
      LOOKUP idx, [0, 1, 2, 1, 0, " "], char
      SEROUT Sio, Baud, [char]
      PAUSE 50
    NEXT
  NEXT

IO_Port_Test:
  DEBUG CRSRXY, 0, 9, "IO Port Test"
  SEROUT Sio, Baud, [LcdI, LcdI, LcdSetIO, %10000000]
```

```
DEBUG CRSRXY, 0, 10, " - port.7 = "
SEROUT Sio, Baud, [LcdI, LcdI, LcdRdIO]
SERIN Sio, Baud, [char]
DEBUG BIN1 char.BIT7,
CRSRXY, 0, 11, " - Writing to port "
FOR idx = 0 TO 6
  char = %00000001 << idx
  SEROUT Sio, Baud, [LcdI, LcdI, LcdWrIO, char]
  SEROUT Sio, Baud, [LcdI, LcdLine2]
  SEROUT Sio, Baud, ["PORT: ", BIN8 char]
  PAUSE 250
NEXT
FOR idx = 5 TO 0
  char = %00000001 << idx
  SEROUT Sio, Baud, [LcdI, LcdI, LcdWrIO, char]
  SEROUT Sio, Baud, [LcdI, LcdLine2]
  SEROUT Sio, Baud, ["PORT: ", BIN8 char]
  PAUSE 250
NEXT

GOTO Main
```

SX/B Example: Sony IRCS Decoding

Examples Index



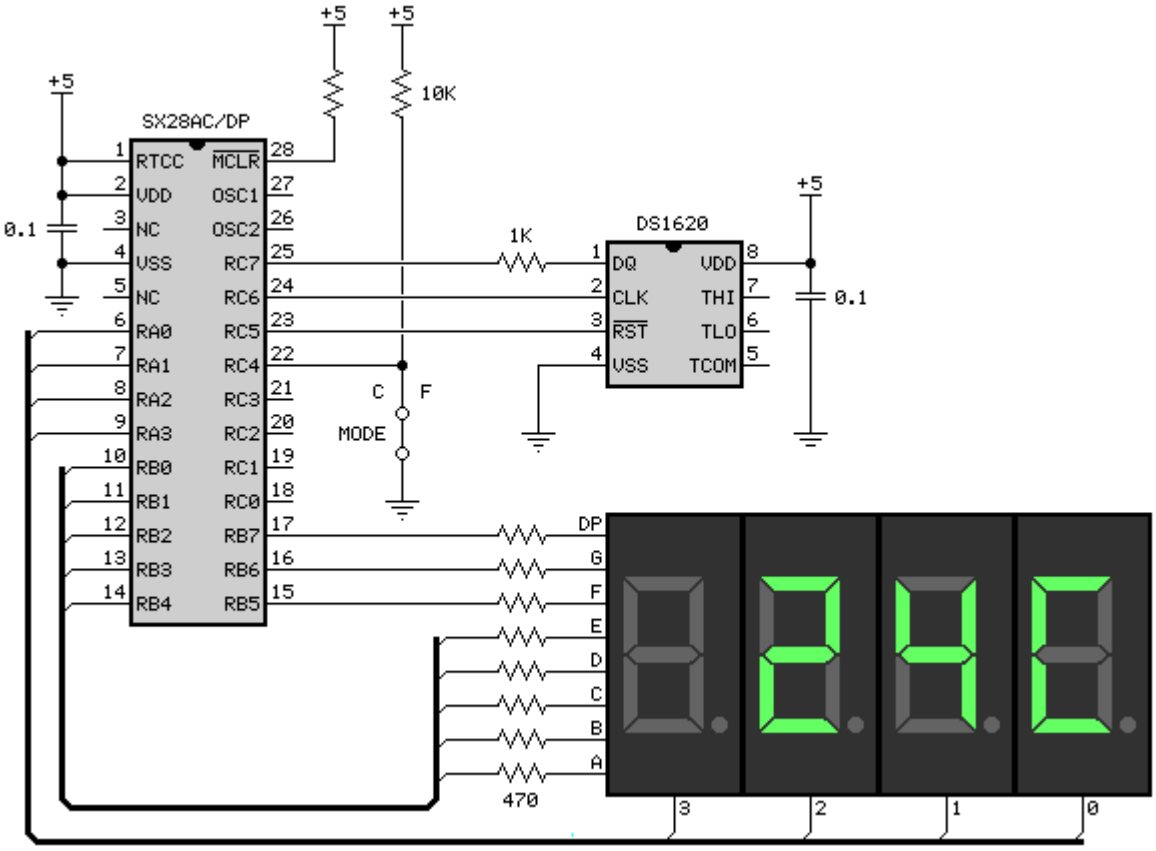

```
' =============================================================================
'
' File...... SIRCS.SXB
' Purpose... Sony IRCS decoder
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =============================================================================

' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Receives and decodes Sony IR Control System stream from IR demodulator.
' Decodes 12 bits, displays lower seven on LEDs.
'
' References (free downloads from www.parallax.com):
' -- "The Nuts & Volts of BASIC Stamps" column 76
' -- "IR Remote for the Boe-Bot" (#28139)

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE      SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "SIRCS"

' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

IR          PIN   RA.0 INPUT        ' IR input
Leds        PIN   RBC OUTPUT        ' LEDs on RB/RC

' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

StartBit    CON   216              ' 90% of 2400 us
```

```
OneBit      CON   108               ' 90% of 1200 us
ZeroBit     CON    54               ' 90% of 600 us

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

irCode      VAR   Word              ' entire code
cmdCode     VAR   irCode_LSB        ' IR command code (7 bits)
devCode     VAR   irCode_MSB        ' IR device code (5 bits)

tmpB1       VAR   Byte              ' subroutine work vars
tmpB2       VAR   Byte
tmpB3       VAR   Byte
tmpW1       VAR   Word

' =======================================================================
 PROGRAM Start
' =======================================================================

' -----------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------

GET_IR_PULSE      FUNC  1           ' get pulse from IR pin
GET_SIRCS         FUNC  2           ' get code from SIRCS

' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:
  PLP_A = %0001                     ' pull-up unused pins
  TRIS_B = %00000000                ' make LED pins outputs
  TRIS_C = %00000000

Main:
  DO
    Leds = GET_SIRCS                ' get and show IR code
  LOOP
  END

' -----------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------

' pulseWidth = GET_IR_PULSE
' -- waits for and measures 1-0-1 pulse on IR pin
' -- return value (0 to 255) is in 10 uS units

FUNC GET_IR_PULSE
  PULSIN IR, 0, tmpB1               ' wait for pulse
  RETURN tmpB1                      ' return to caller
ENDFUNC

' -----------------------------------------------------------------------

' GET_SIRCS
' -- waits for Sony IRCS input stream
' -- puts 7-bit command into LSB of return value
```

```
' -- puts 5-bit device code into MSB of return value

FUNC GET_SIRCS
  tmpW1 = 0                          ' clear working output
  DO
    tmpB2 = GET_IR_PULSE             ' get bit from IR
  LOOP UNTIL tmpB2 >= StartBit       ' loop until valid start bit
  FOR tmpB3 = 0 TO 6                 ' get 7-bit code
    tmpW1_LSB = tmpW1_LSB >> 1       ' prep for LSB
    tmpB2 = GET_IR_PULSE             ' measure data bit
    IF tmpB2 >= OneBit THEN          ' valid "1" bit
      tmpW1_LSB.6 = 1                ' yes, set the bit
    ENDIF
  NEXT
  FOR tmpB3 = 0 TO 4                 ' get 5-bit code
    tmpW1_MSB = tmpW1_MSB >> 1       ' prep for LSB
    tmpB2 = GET_IR_PULSE             ' measure data bit
    IF tmpB2 >= OneBit THEN          ' valid "1" bit
      tmpW1_MSB.4 = 1                ' yes, set the bit
    ENDIF
  NEXT
  RETURN tmpW1
ENDFUNC
```

SX/B Example: Thermometer

Examples Index




```
' ===============================================================================
'
' File..... THERMOMETER.SXB
' Purpose... Dual-mode digital thermometer
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' ===============================================================================
' -------------------------------------------------------------------------------
' Program Description
' -------------------------------------------------------------------------------
'
' Displays the temperature from a DS1620 thermo chip on a multiplexed 7-
' segment display. The display is multiplexed by interrupt code so that
' no additional components are required. A mode input allows the temper-
' ature to be displayed in Celsius or Fahrenheit.
'
' When the temperature falls below 0C (32F) "-Lo-" is displayed. When the
' temp is above 49C (120 F) "-Hi-" is displayed.

' -------------------------------------------------------------------------------
' Device Settings
' -------------------------------------------------------------------------------
```

```
DEVICE            SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "THERMO"

' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

DQ          PIN   RC.7              ' DS1620.1
Clock       PIN   RC.6              ' DS1620.2
Rst         PIN   RC.5              ' DS1620.3
DispMode    PIN   RC.4              ' display mode, C or F
Segs        PIN   RB                ' display segments
DigCtrl     PIN   RA                ' digit control (cathode)

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

Blank       CON   %00000000         ' blank display
Dash        CON   %01000000         ' pattern for "-"
Ltr_C       CON   %00111001         ' pattern for "C"
Ltr_F       CON   %01110001         ' pattern for "F"
Ltr_L       CON   %00111000         ' pattern for "L"
Ltr_o       CON   %01011100         ' pattern for "o"
Ltr_H       CON   %01110110         ' pattern for "H"
Ltr_i       CON   %00010000         ' pattern for "i"

TC          CON   0                 ' mode = Celsius
TF          CON   1                 ' mode = Fahrenheit

' *** DS1620 Commands ***

RdTmp       CON   $AA               ' read temperature
WrHi        CON   $01               ' write TH (high temp)
WrLo        CON   $02               ' write TL (low temp)
RdHi        CON   $A1               ' read TH
RdLo        CON   $A2               ' read TL
StartC      CON   $EE               ' start conversion
StopC       CON   $22               ' stop conversion
WrCfg       CON   $0C               ' write config register
RdCfg       CON   $AC               ' read config register

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

display     VAR   Byte(4)           ' multiplexed segments
digPntr     VAR   Byte              ' digit pointer
digLimit    VAR   Byte              ' 0 - 3
theTemp     VAR   Byte              ' temperature
tSign       VAR   Byte              ' sign in bit 0
work        VAR   Byte

tmpB1       VAR   Byte              ' subroutine work vars
tmpB2       VAR   Byte
tmpW1       VAR   Word

' =======================================================================
```

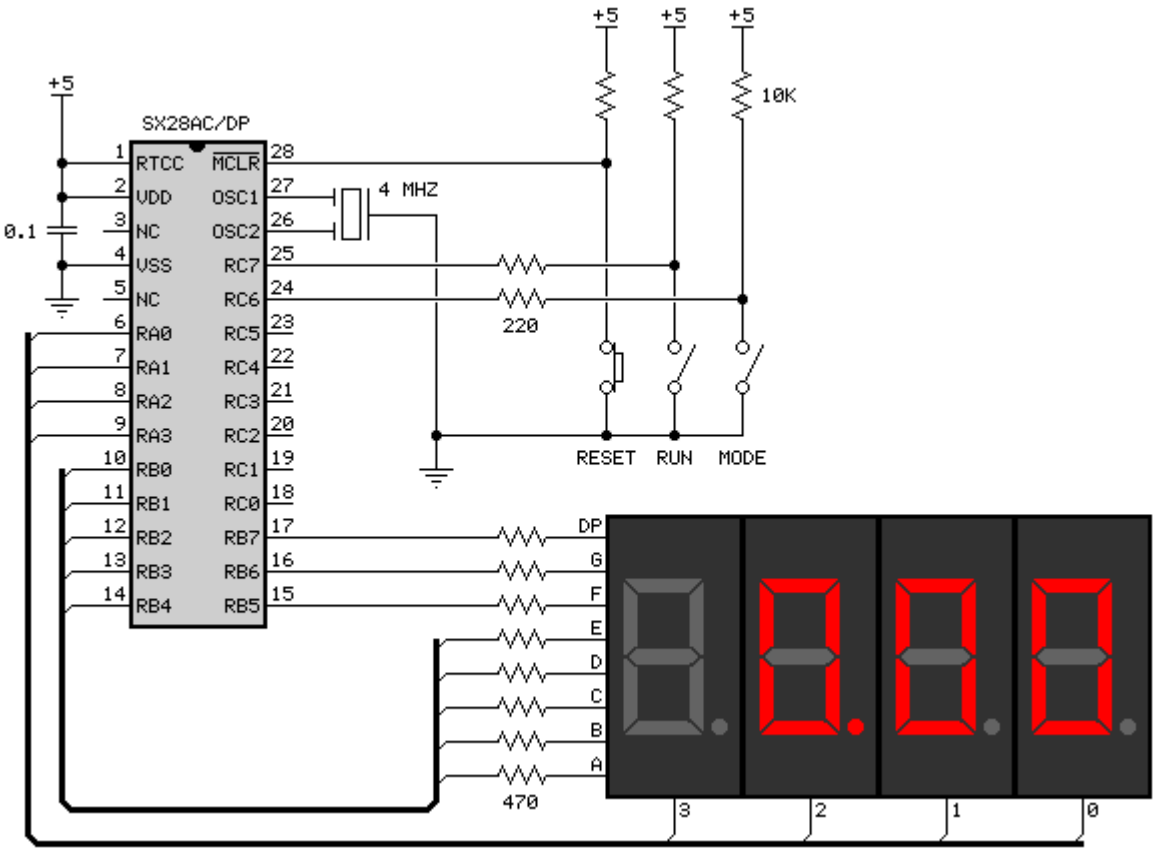
```
 INTERRUPT 200
' ============================================================================

' Points to next digit of display every 5 milliseconds (200 times/sec).

ISR_Start:
  INC digPntr                             ' point to next digit
  IF digPntr < digLimit THEN Update_Segs  ' update digit
  digPntr = 0                             ' wrap if needed

Update_Segs:
  Segs = Blank                            ' blank segs
  READ DigMap + digPntr, DigCtrl          ' select display element
  Segs = display(digPntr)                 ' output new digit segs

ISR_Exit:
  RETURNINT

' ============================================================================
 PROGRAM Start
' ============================================================================

' ----------------------------------------------------------------------------
' Subroutines
' ----------------------------------------------------------------------------

INIT_1620   SUB   0                       ' initialize DS1620
RD_1620     FUNC  1                       ' get temp from DS1620
NEW_DISPLAY SUB   2                       ' update display

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:
  DigCtrl = %1111                         ' disable all digits
  TRIS_A = %0000                          ' make dig pins outputs
  Segs = Blank                            ' clear seg drivers
  TRIS_B = %00000000                      ' make seg pins outputs
  TRIS_C = %10011111                      ' clock and rst are outputs
  INIT_1620
  PUT display(0), Blank, Blank, Blank, Blank    ' clear display
  digLimit = 4                            ' use all digits (1 - 4)
  digPntr = 3

Main:
  DO
    theTemp = RD_1620                     ' read temperature
    NEW_DISPLAY theTemp, tSign            ' display value
    PAUSE 500                             ' delay between reads
  LOOP

' ----------------------------------------------------------------------------
' Subroutine Code
' ----------------------------------------------------------------------------

' Initialize DS1620 for free-run mode and for use with a host CPU

SUB INIT_1620
  Rst = 1                                 ' select device
```

```
  SHIFTOUT DQ, Clock, LSBFIRST, WrCfg           ' write to config register
  SHIFTOUT DQ, Clock, LSBFIRST, %00000010       ' with CPU; free-run
  Rst = 0                                   ' deselect device
  PAUSE 10                                  ' allow DS1620 EE to write
  Rst = 1                                   ' reselect
  SHIFTOUT DQ, Clock, LSBFIRST, StartC      ' start conversion
  Rst = 0                                   ' deselect
ENDSUB

' -----------------------------------------------------------------------

' Use: theTemp = RD1620
' -- returns temp in whole degrees C in 'theTemp'

FUNC RD_1620
  Rst = 1                                   ' select device
  SHIFTOUT DQ, Clock, LSBFIRST, RdTmp       ' send read temp command
  SHIFTIN DQ, Clock, LSBPRE, tmpB1          ' get temp (C x 0.5)
  SHIFTIN DQ, Clock, LSBPRE, tSign\1        ' get sign bit
  Rst = 0                                   ' deselect device
  tmpB1 = tmpB1 + tmpB1.0                   ' round up
  tmpB1 = tmpB1 >> 1                        ' remove half bit
  RETURN tmpB1
ENDFUNC

' -----------------------------------------------------------------------

' Use: NEW_DISPLAY temperature, sign
' -- puts 'temperature' in display (pass value in degrees C)
' -- display mode (C or F) controlled by RC.4 input

SUB NEW_DISPLAY
  tmpB1 = __PARAM1                          ' temperature
  tmpB2 = __PARAM2                          ' sign bit
  IF tmpB2 = 1 THEN                         ' negative temp?
    digLimit = 4
    PUT display(0), Dash, Ltr_o, Ltr_L, Dash    ' show "-Lo-"
    GOTO Display_Done
  ENDIF
  IF tmpB1 > 49 THEN                        ' too high?
    digLimit = 4
    PUT display(0), Dash, Ltr_i, Ltr_H, Dash    ' show "-Hi-"
    GOTO Display_Done
  ENDIF

Check_Mode:
  IF DispMode = TC THEN                     ' check mode input switch
    display(0) = Ltr_C
  ELSE
    display(0) = Ltr_F
    tmpW1 = tmpB1 */ $1CC                   ' temp x 1.8 (9/5)
    tmpB1 = tmpW1_LSB + 32
  ENDIF

Set_Display:                                ' blank leading zeros
  digLimit = 2                              ' 1-digit temp
  IF tmpB1 < 10 THEN Show_Temp
  INC digLimit                              ' 2-digit temp
  IF tmpB1 < 100 THEN Show_Temp
  INC digLimit                              ' 3-digit temp
```

```
Show_Temp:
  tmpB2 = tmpB1 / 100                     ' get hundreds digit
  tmpB1 = __REMAINDER                     ' save 10's and 1's
  READ SegMap + tmpB2, display(3)         ' get segment map 100's
  tmpB2 = tmpB1 / 10                      ' get 10's digit
  tmpB1 = __REMAINDER                     ' save 1's
  READ SegMap + tmpB2, display(2)         ' get segment map for 10's
  READ SegMap + tmpB1, display(1)         ' get segment map for 1's

Display_Done:
ENDSUB

' ============================================================================
' User Data
' ============================================================================

SegMap: ' segments maps
'       .gfedcba
  DATA %00111111                    ' 0
  DATA %00000110                    ' 1
  DATA %01011011                    ' 2
  DATA %01001111                    ' 3
  DATA %01100110                    ' 4
  DATA %01101101                    ' 5
  DATA %01111101                    ' 6
  DATA %00000111                    ' 7
  DATA %01111111                    ' 8
  DATA %01100111                    ' 9

DigMap:                             ' digit select map
  DATA %11111110
  DATA %11111101
  DATA %11111011
  DATA %11110111
```

SX/B Example: Clock / Timer

Examples Index




```
' =============================================================================
'
' File...... TIMER.SXB
' Purpose... Dual-Mode digital timer
' Author.... (c) Parallax, Inc. -- All Rights Reserved
' E-mail.... support@parallax.com
' Started...
' Updated... 05 JUL 2006
'
' =============================================================================
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Displays running clock/timer on multiplexed 7-segment display. The
' four-digit display is multiplexed through a "virtual" driver which is
' handled in the INTERRUPT routine.

' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE            SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ        4_000_000
ID          "TIMER"
```

```
' -----------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------

DigCtrl     PIN   RA                ' digit control (cathodes)
Segs        PIN   RB                ' display segments (anodes)
TmrMode     PIN   RC.6               ' mode: 0 = mmss, 1 = hhmm
TmrEnable   PIN   RC.7               ' enable input, 1 = run

' -----------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------

Yes         CON   1
No          CON   0
MaxDigit    CON   4                 ' 4-digit display
DecPnt      CON   %10000000         ' decimal point mask
TmrMMSS     CON   1                 ' show MM.SS (no blink)
TmrHHMM     CON   0                 ' show HH.MM (blink DP)
TmrRun      CON   1                 ' run timer
TmrHold     CON   0                 ' hold timer
MaxHr       CON   24                ' 0 .. 23 (clock mode)
                                    ' make 100 for 0 .. 99

' -----------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------

ms          VAR   Word              ' milliseconds
clock       VAR   Byte(3)           ' clock array
secs        VAR   clock(0)          ' seconds
mins        VAR   clock(1)          ' minutes
hrs         VAR   clock(2)
blink       VAR   secs.0            ' DP blink control bit
display     VAR   Byte(MaxDigit)    ' multiplexed segments
digPntr     VAR   Byte              ' digit pointer
tmpB1       VAR   Byte
tmpB2       VAR   Byte

' -----------------------------------------------------------------------
 INTERRUPT 1000
' -----------------------------------------------------------------------

' The ISR is called every millisecond using the Rate parameter of the
' INTERRUPT instruction. With a 4 MHz clock, the prescaler will be set
' 1:16 and the ISR will run every 250 RTCC cycles.
'
' If the timer is enabled (TmrEnable pin = 1), the timer values will be
' updated every millisecond, otherwise only the display will be refreshed
' (one digit per interrupt).

ISR_Start:
  IF TmrEnable = TmrHold THEN Next_Digit  ' skip clock update if 0
Update_Timer:
  INC ms ' update ms counter
  IF ms = 1000 THEN                     ' check for 1 second
    ms = 0
    INC secs
    IF secs = 60 THEN                   ' check for new minute
```

```
      secs = 0
      INC mins
      IF mins = 60 THEN                   ' check for new hour
        mins = 0
        INC hrs
        IF hrs = MaxHr THEN
          hrs = 0
        ENDIF
      ENDIF
    ENDIF
  ENDIF

Next_Digit:
  INC digPntr                             ' point to next digit
  IF digPntr = MaxDigit THEN              ' check pointer
    digPntr = 0                           ' wrap if needed
   NDIF

Update_Segs:
  Segs = %00000000                        ' blank segments
  READ DigMap + digPntr, DigCtrl          ' update digit control
  Segs = display(digPntr)                 ' output new segments

ISR_Exit:
  RETURNINT

' ============================================================================
 PROGRAM Start
' ============================================================================

' ----------------------------------------------------------------------------
' Subroutine Declarations
' ----------------------------------------------------------------------------

CLOCK_MMSS  SUB   0                       ' show mins & secs
CLOCK_HHMM  SUB   0                       ' show hrs & mins

' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Start:
  DigCtrl = %1111                         ' disable all digits
  TRIS_A = %0000                          ' make dig pins outputs
  Segs = %00000000                        ' clear seg drivers
  TRIS_B = %00000000                      ' make seg pins outputs
  PLP_C = %11000000                       ' pull-up unused pins

Main:
  DO
    PAUSE 50
    IF TmrMode = TmrHHMM THEN ' check mode
      CLOCK_HHMM
    ELSE
      CLOCK_MMSS
    ENDIF
  LOOP

' ----------------------------------------------------------------------------
' Subroutine Code
```

```
' -----------------------------------------------------------------------

' Display timer in MM.SS (00.00 .. 59.59) format with solid DP

SUB CLOCK_MMSS                          ' display mins & secs
  tmpB1 = mins / 10                     ' get 10's digit
  tmpB2 = __REMAINDER                   ' save 1's digit
  READ SegMap + tmpB1, display(3)
  READ SegMap + tmpB2, display(2)
  display(2) = display(2) | DecPnt      ' add DP to hr01 digit
  tmpB1 = secs / 10
  tmpB2 = __REMAINDER
  READ SegMap + tmpB1, display(1)
  READ SegMap + tmpB2, display(0)
ENDSUB

' -----------------------------------------------------------------------

' Display timer in HH.MM (00.00 .. 23.59) format with blinking DP

CLOCK_HHMM:                             ' display hours & mins
  tmpB1 = hrs / 10
  tmpB2 = __REMAINDER
  READ SegMap + tmpB1, display(3)
  READ SegMap + tmpB2, display(2)
  IF blink = Yes THEN
    display(2) = display(2) | DecPnt    ' blink DP on hr01 digit
  ENDIF
  tmpB1 = mins / 10
  tmpB2 = __REMAINDER
  READ SegMap + tmpB1, display(1)
  READ SegMap + tmpB2, display(0)
ENDSUB

' =======================================================================
' User Data
' =======================================================================

SegMap: ' segments maps
  '     .gfedcba
  DATA %00111111                  ' 0
  DATA %00000110                  ' 1
  DATA %01011011                  ' 2
  DATA %01001111                  ' 3
  DATA %01100110                  ' 4
  DATA %01101101                  ' 5
  DATA %01111101                  ' 6
  DATA %00000111                  ' 7
  DATA %01111111                  ' 8
  DATA %01100111                  ' 9

DigMap: ' digit select map
  DATA %11111110
  DATA %11111101
  DATA %11111011
  DATA %11110111
```