SPC

Version 0.9.5

Generated by Doxygen 1.8.2

Wed Feb 20 2013 17:31:20

Contents

1 SPC Programmer's Guide

February 20, 2013

by John Hansen

- Introduction
- The SPC Language

2 Introduction

SPC stands for SuperPro C.

It is a simple language for programming the HiTechnic SuperPro prototyping sensor board. The SuperPro has a bytecode interpreter which can be used to execute programs. The SPC compiler translates a source program into SuperPro bytecodes, which can then be executed on the target itself. Although the preprocessor and control structures of SPC are very similar to C, SPC is not a general-purpose programming language - there are many restrictions that stem from limitations of the SuperPro bytecode interpreter.

Logically, SPC is defined as two separate pieces. The SPC language describes the syntax to be used in writing programs. The SPC Application Programming Interface (API) describes the system functions, constants, and macros that can be used by programs. This API is defined in a special file known as a "header file" which is, by default, automatically included when compiling a program.

This document describes both the SPC language and the SPC API. In short, it provides the information needed to write SPC programs. Since there are different interfaces for SPC, this document does not describe how to use any specific

SPC implementation (such as the command-line compiler or Bricx Command Center). Refer to the documentation provided with the SPC tool, such as the SPC User Manual, for information specific to that implementation.

For up-to-date information and documentation for SPC, visit the SPC website at http://bricxcc.-sourceforge.net/spc/.

3 The SPC Language

This section describes the SPC language.

This includes the lexical rules used by the compiler, the structure of programs, statements and expressions, and the operation of the preprocessor.

SPC is a case-sensitive language, just like C and C++, which means the identifier "xYz" is not the same identifier as "Xyz". Similarly, the "if" statement begins with the keyword "if" but "iF", "If", or "IF" are all just valid identifiers - not keywords.

- Lexical Rules
- Program Structure
- Statements
- Expressions
- The Preprocessor

3.1 Lexical Rules

The lexical rules describe how SPC breaks a source file into individual tokens.

This includes the way comments are written, the handling of whitespace, and valid characters for identifiers.

- Comments
- Whitespace
- Numerical Constants
- String Constants
- Character Constants
- System Constants
- Identifiers and Keywords

3.1.1 Comments

Two forms of comments are supported in SPC.

The first are traditional C comments. They begin with '/*' and end with '* /'. These comments are allowed to span multiple lines, but they cannot be nested.

```
/* this is a comment */

/* this is a two
   line comment */

/* another comment...
   /* trying to nest...
      ending the inner comment...*/
   this text is no longer a comment! */
```

The second form of comments supported in SPC begins with '//' and continues to the end of the current line. These are sometimes known as C++ style comments.

```
// a single line comment
```

As you might guess, the compiler ignores comments. Their only purpose is to allow the programmer to document the source code.

3.1.2 Whitespace

Whitespace consists of all spaces, tabs, and newlines.

It is used to separate tokens and to make a program more readable. As long as the tokens are distinguishable, adding or subtracting whitespace has no effect on the meaning of a program. For example, the following lines of code both have the same meaning:

```
x=2;
x   =  2  ;
```

Some of the C++ operators consist of multiple characters. In order to preserve these tokens, whitespace cannot appear within them. In the example below, the first line uses a right shift operator ('>>'), but in the second line the added space causes the '>' symbols to be interpreted as two separate tokens and thus results in a compiler error.

```
x = 1 >> 4; // set x to 1 right shifted by 4 bits
x = 1 > > 4; // error
```

3.1.3 Numerical Constants

Numerical constants may be written in either decimal or hexadecimal form.

Decimal constants consist of one or more decimal digits. Decimal constants may optionally include a decimal point along with one or more decimal digits following the decimal point. Hexadecimal constants start with 0x or 0X followed by one or more hexadecimal digits.

```
x = 10;  // set x to 10
x = 0x10; // set x to 16 (10 hex)
f = 10.5; // set f to 10.5
```

3.1.4 String Constants

String constants in SPC, just as in C, are delimited with double quote characters.

String constants can only be used in a few API functions that require a const char * input parameter.

```
puts("testing\n");
printf("testing %d\n", value);
```

3.1.5 Character Constants

Character constants in SPC are delimited with single quote characters and may contain a single ASCII character.

The value of a character constant is the numeric ASCII value of the character.

```
char ch = 'a'; // ch == 97
```

3.1.6 System Constants

In SPC you can define special system memory address constants that are treated like a variable with an absolute memory address.

A system address is simply a numeric constant preceded by the '@' symbol.

```
int volt = @0x00; // read the voltage from analog input A0.
@0x0C = 1000; // set countdown timer 0 to 1000.
```

System constants can also be used to read and write to an area of shared memory that can be accessed via I2C from a device such as the NXT. The shared memory area from 0x20 - 0x3F is mapped to the byte addressed I2C 0x80 - 0xFF memory address range. This permits data to be exchanged between an attached NXT and the user program. See the Shared Memory page for predefined constants to use this area of memory.

3.1.7 Identifiers and Keywords

Identifiers are used for variable, task, function, and subroutine names.

The first character of an identifier must be an upper or lower case letter or the underscore ('_'). Remaining characters may be letters, numbers, and underscores.

A number of tokens are reserved for use in the SPC language itself. These are called keywords and may not be used as identifiers. A complete list of keywords appears below:

- The asm statement
- bool
- The break statement
- The case label
- char
- const
- The continue statement
- The default label
- The do statement
- The if-else statement
- enum
- The false condition
- The for statement

- The goto statement
- The if statement
- The inline keyword
- int
- long
- The repeat statement
- The return statement
- The start statement
- static
- Structures
- The sub keyword
- The switch statement
- Tasks
- The true condition
- typedef
- The until statement
- The void keyword
- The while statement

3.1.7.1 const

The const keyword is used to alter a variable declaration so that the variable cannot have its value changed after it is initialized.

The initialization must occur at the point of the variable declaration.

```
const int myConst = 23; // declare and initialize constant integer
task main() {
  int x = myConst; // this works fine
  myConst++; // compiler error - you cannot modify a constant's value
}
```

3.1.7.2 enum

The enum keyword is used to create an enumerated type named name.

The syntax is show below.

```
enum [name] {name-list} var-list;
```

The enumerated type consists of the elements in name-list. The var-list argument is optional, and can be used to create instances of the type along with the declaration. For example, the following code creates an enumerated type for colors:

```
enum ColorT {red, orange, yellow, green, blue, indigo, violet};
```

In the above example, the effect of the enumeration is to introduce several new constants named red, orange, yellow, etc. By default, these constants are assigned consecutive integer values starting at zero. You can change the values of those constants, as shown by the next example:

```
enum ColorT { red = 10, blue = 15, green };
```

In the above example, green has a value of 16. Once you have defined an enumerated type you can use it to declare variables just like you use any native type. Here are a few examples of using the enum keyword:

```
// values start from 0 and increment upward by 1
enum { ONE, TWO, THREE };
// optional equal sign with constant expression for the value
enum { SMALL=10, MEDIUM=100, LARGE=1000 };
// names without equal sign increment by one from last name's value
enum { FRED=1, WILMA, BARNEY, BETTY };
// optional named type (like a typedef)
enum TheSeasons { SPRING, SUMMER, FALL, WINTER };
// optional variable at end
enum Days {
   saturday,             // saturday = 0 by default
   sunday = 0x0,           // sunday = 0 as well
   monday,               // monday = 1
   tuesday,              // tuesday = 2
   wednesday,            // etc.
   thursday,
   friday
} today;                 // Variable today has type Days

Days tomorrow;

task main()
{
  TheSeasons test = FALL;
  today = monday;
  tomorrow = today+1;
  printf("%d\n", THREE);
  printf("%d\n", MEDIUM);
  printf("%d\n", FRED);
  printf("%d\n", SPRING);
  printf("%d\n", friday);
  printf("%d\n", today);
  printf("%d\n", test);
  printf("%d\n", tomorrow);
  Wait(SEC_5);
}
```

3.1.7.3 static

The static keyword is used to alter a variable declaration so that the variable is allocated statically - the lifetime of the variable extends across the entire run of the program - while having the same scope as variables declared without the static keyword.

Note that the initialization of automatic and static variables is quite different. Automatic variables (local variables are automatic by default, unless you explicitly use static keyword) are initialized during the run-time, so the initialization will be executed whenever it is encountered in the program. Static (and global) variables are initialized during the compile-time, so the initial values will simply be embeded in the executable file itself.

```
void func() {
  static int x = 0; // x is initialized only once across three calls of func()
  NumOut(0, LCD_LINE1, x); // outputs the value of x
  x = x + 1;
}

task main() {
  func(); // prints 0
  func(); // prints 1
  func(); // prints 2
}
```

3.1.7.4 typedef

A typedef declaration introduces a name that, within its scope, becomes a synonym for the type given by the type-declaration portion of the declaration.

```
typedef type-declaration synonym;
```

You can use typedef declarations to construct shorter or more meaningful names for types already defined by the language or for types that you have declared. Typedef names allow you to encapsulate implementation details that may change.

A typedef declaration does not introduce a new type - it introduces a new name for an existing type. Here are a few examples of how to use the typedef keyword:

```
typedef char FlagType;
const FlagType x;
typedef char CHAR;          // Character type.
CHAR ch;
```

3.2 Program Structure

An SPC program is composed of code blocks and variables.

There are two distinct types of code blocks: tasks and functions. Each type of code block has its own unique features, but they share a common structure.

- Code Order
- Tasks
- Functions
- Variables
- Structures
- Arrays

3.2.1 Code Order

Code order has two aspects: the order in which the code appears in the source code file and the order in which it is executed at runtime.

The first will be referred to as the lexical order and the second as the runtime order.

The lexical order is important to the SPC compiler, but not to the SuperPro brick. This means that the order in which you write your task and function definitions has no effect on the runtime order. The rules controlling runtime order are:

1. There must be a task called main and this task will always run first.
2. The time at which any other task will run is determined by the placement of API functions and keywords that start other tasks.
3. A function will run whenever it is called from another block of code.

This last rule may seem trivial, but it has important consequences when multiple tasks are running. If a task calls a function that is already in the midst of running because it was called first by another task, unpredictable behavior and results may ensue. Tasks can share functions by treating them as shared resources and using code to prevent one task from calling the function while another task is using it.

The rules for lexical ordering are:

1. Any identifier naming a task or function must be known to the compiler before it is used in a code block.
2. A task or function definition makes its naming identifier known to the compiler.
3. A task or function declaration also makes a naming identifier known to the compiler.
4. Once a task or function is defined it cannot be redefined or declared.
5. Once a task or function is declared it cannot be redeclared.

Sometimes you will run into situations where is impossible or inconvenient to order the task and function definitions so the compiler knows every task or function name before it sees that name used in a code block. You can work around this by inserting task or function declarations of the form

task *name*();

return_type name(*argument_list*);

before the code block where the first usage occurs. The *argument_list* must match the list of formal arguments given later in the function's actual definition.

3.2.2 Tasks

Since the SuperPro supports multi-threading, a task in SPC directly corresponds to a SuperPro thread or process.

Tasks are defined using the task keyword with the syntax shown in the code sample below.

```
task name()
{
  // the task's code is placed here
}
```

The name of the task may be any legal identifier. A program must always have at least one task - named "main" - which is started whenever the program is run. The body of a task consists of a list of statements.

You can start tasks with the start statement, which is discussed below.

The StopAllTasks API function stops all currently running tasks. You can also stop all tasks using the Stop function. A task can stop itself via the ExitTo function. Finally, a task will stop itself simply by reaching the end of its body.

3.2.3 Functions

It is often helpful to group a set of statements together into a single function, which your code can then call as needed.

SPC supports functions with arguments and return values. Functions are defined using the syntax below.

```
[inline] return_type name(argument_list)
{
        // body of the function
}
```

The return type is the type of data returned. In the C programming language, functions must specify the type of data they return. Functions that do not return data simply return void.

Additional details about the keywords inline, and void can be found below.

- The inline keyword
- The void keyword

The argument list of a function may be empty, or may contain one or more argument definitions. An argument is defined by a type followed by a name. Commas separate multiple arguments. All values are represented as bool, char, int, long, struct types, or arrays of any type.

SPC supports specifying a default value for function arguments that are not struct or array types. Simply add an equal sign followed by the default value. Specifying a default value makes the argument optional when you call the function. All optional arguments must be at the end of the argument list.

```
int foo(int x, int y = 20)
{
	return x*y;
}

task main()
{
	printf("%d\n", foo(10)); outputs 200
	printf("%d\n", foo(10, 5)); outputs 50
	Wait(SEC_10); // wait 10 seconds
}
```

SPC also supports passing arguments by value, by constant value, by reference, and by constant reference. These four modes for passing parameters into a function are discussed below.

When arguments are passed by value from the calling function or task to the called function the compiler must allocate a temporary variable to hold the argument. There are no restrictions on the type of value that may be used. However, since the function is working with a copy of the actual argument, the caller will not see any changes the called function makes to the value. In the example below, the function foo attempts to set the value of its argument to 2. This is perfectly legal, but since foo is working on a copy of the original argument, the variable y from the main task remains unchanged.

```
void foo(int x)
{
	x = 2;
}

task main()
{
	int y = 1;	// y is now equal to 1
	foo(y); // y is still equal to 1!
}
```

The second type of argument, const arg_type, is also passed by value. If the function is an inline function then arguments of this kind can sometimes be treated by the compiler as true constant values and can be evaluated at compile-time. If the function is not inline then the compiler treats the argument as if it were a constant reference, allowing you to pass either constants or variables. Being able to fully evaluate function arguments at compile-time can be important since some SPC API functions only work with true constant arguments.

```
void foo(const int x)
{
	x = 1;  // error - cannot modify argument
	Wait(SEC_1);
}

task main()
{
	int x = 5;
	foo(5); // ok
	foo(4*5);	// expression is still constant
	foo(x); // x is not a constant but is okay
}
```

The third type, arg_type &, passes arguments by reference rather than by value. This allows the called function to modify the value and have those changes be available in the calling function after the called function returns. However, only variables may be used when calling a function using arg_type & arguments:

```
void foo(int &x)
{
        x = 2;
}

task main()
{
        int y = 1;      // y is equal to 1

        foo(y); // y is now equal to 2
        foo(2); // error - only variables allowed
}
```

The fourth type, const arg_type &, is interesting. It is also passed by reference, but with the restriction that the called function is not allowed to modify the value. Because of this restriction, the compiler is able to pass anything, not just variables, to functions using this type of argument. Currently, passing an argument by reference in SPC is not as optimal as it is in C. A copy of the argument is still made but the compiler will enforce the restriction that the value may not be modified inside the called function.

Functions must be invoked with the correct number and type of arguments. The code example below shows several different legal and illegal calls to function foo.

```
void foo(int bar, const int baz)
{
        // do something here...
}

task main()
{
        int x;  // declare variable x
        foo(1, 2);      // ok
        foo(x, 2);      // ok
        foo(2); // error - wrong number of arguments!
}
```

3.2.3.1 The inline keyword

You can optionally mark SPC functions as inline functions.

This means that each call to the function will create another copy of the function's code. Unless used judiciously, inline functions can lead to excessive code size.

If a function is not marked as inline then an actual SuperPro subroutine is created and the call to the function in SPC code will result in a subroutine call to the SuperPro subroutine. The total number of non-inline functions (aka subroutines) and tasks must not exceed 256.

The code example below shows how you can use the inline keyword to make a function emit its code at the point where it is called rather than requiring a subroutine call.

```
inline void foo(int value)
{
  Wait(value);
}

task main()
{
  foo(MS_100);
  foo(MS_10);
  foo(SEC_1);
  foo(MS_50);
}
```

In this case task main will contain 4 Wait calls rather than 4 calls to the foo subroutine since it was expanded inline.

3.2.3.2 The void keyword

The void keyword allows you to define a function that returns no data.

Functions that do not return any value are sometimes referred to as procedures or subroutines. The sub keyword is an alias for void. Both of these keywords can only be used when declaring or defining a function. Unlike C you cannot use void when declaring a variable type.

In NQC the void keyword was used to declare inline functions that could have arguments but could not return a value. In SPC void functions are not automatically inline as they were in NQC. To make a function inline you have to use the inline keyword prior to the function return type as described in the Functions section above.

- The sub keyword

3.2.3.2.1 The sub keyword

The sub keyword allows you to define a function that returns no data.

Functions that do not return any value are sometimes referred to as procedures or subroutines. The sub keyword is an alias for void. Both of these keywords can only be used when declaring or defining a function.

In NQC you used this keyword to define a true subroutine which could have no arguments and return no value. For the sake of C compatibility it is preferrable to use the void keyword if you want to define a function that does not return a value.

3.2.4 Variables

All variables in SPC are defined using one of the types listed below:

- bool
- char
- int
- long
- Structures
- Arrays

Variables are declared using the keyword(s) for the desired type, followed by a comma-separated list of variable names and terminated by a semicolon (';'). Optionally, an initial value for each variable may be specified using an equals sign ('=') after the variable name. Several examples appear below:

```
int x;          // declare x
bool y,z;       // declare y and z
long a=1,b;     // declare a and b, initialize a to 1
int data[10]; // an array of 10 zeros in data
bool flags[] = {true, true, false, false};
```

Global variables are declared at the program scope (outside of any code block). Once declared, they may be used within all tasks, functions, and subroutines. Their scope begins at declaration and ends at the end of the program.

Local variables may be declared within tasks and functions. Such variables are only accessible within the code block in which they are defined. Specifically, their scope begins with their declaration and ends at the end of their code block. In the case of local variables, a compound statement (a group of statements bracketed by '{' and '}') is considered a block:

```
int x;  // x is global

task main()
{
        int y;  // y is local to task main
```

```
        x = y; // ok
        {       // begin compound statement
                int z;  // local z declared
                y = z; // ok
        }
        y = z; // error - z no longer in scope
}

task foo()
{
        x = 1; // ok
        y = 2; // error - y is not global
}
```

3.2.4.1 bool

In SPC the bool type is a signed 32-bit value.

Normally you would only store a zero or one in a variable of this type.

```
bool flag=true;
```

3.2.4.2 char

In SPC the char type is a signed 32-bit value.

The char type is often used to store the ASCII value of a single character. Use Character Constants page has more details about this usage.

```
char ch=12;
char test = 'A';
```

3.2.4.3 int

In SPC the int type is a signed 32-bit value.

This type can store values from INT_MIN to INT_MAX.

```
int x = 0xfff;
int y = -23;
```

3.2.4.4 long

In SPC the long type is a signed 32-bit value.

This type can store values from LONG_MIN to LONG_MAX.

```
long x = 2147000000;
long y = -88235;
```

3.2.5 Structures

SPC supports user-defined aggregate types known as structs.

These are declared very much like you declare structs in a C program.

```
struct car
{
  int car_type;
  int manu_year;
};

struct person
```

```
{
  int age;
  car vehicle;
};

person myPerson;
```

After you have defined the structure type you can use the new type to declare a variable or nested within another structure type declaration. Members (or fields) within the struct are accessed using a dot notation.

```
myPerson.age = 40;
anotherPerson = myPerson;
fooBar.car_type = honda;
fooBar.manu_year = anotherPerson.age;
```

You can assign structs of the same type but the compiler will complain if the types do not match.

3.2.6 Arrays

SPC also support arrays.

Arrays are declared the same way as ordinary variables, but with an open and close bracket following the variable name. Arrays must either have a non-empty size declaration or an initializer following the declaration.

```
int my_array[3];  // declare an array with 3 elements
```

To declare arrays with more than one dimension simply add more pairs of square brackets. The maximum number of dimensions supported in SPC is 4.

```
bool my_array[3][3];  // declare a 2-dimensional array
```

Arrays of up to two dimensions may be initialized at the point of declaration using the following syntax:

```
int X[] = {1, 2, 3, 4}, Y[]={10, 10}; // 2 arrays
int matrix[][] = {{1, 2, 3}, {4, 5, 6}};
```

The elements of an array are identified by their position within the array (called an index). The first element has an index of 0, the second has index 1, and so on. For example:

```
my_array[0] = 123; // set first element to 123
my_array[1] = my_array[2]; // copy third into second
```

SPC also supports specifying an initial size for both global and local arrays. The compiler automatically generates the required code to correctly initialize the array to zeros. If an array declaration includes both a size and a set of initial values the size is ignored in favor of the specified values.

```
task main()
{
  int myArray[10][10];
  int myVector[10];

}
```

3.3 Statements

The body of a code block (task or function) is composed of statements.

Statements are terminated with a semi-colon (';'), as you have seen in the example code above.

- Variable Declaration
- Assignment
- Control Structures
- The asm statement
- Other SPC Statements

3.3.1 Variable Declaration

Variable declaration, which has already been discussed, is one type of statement.

Its purpose is to declare a local variable (with optional initialization) for use within the code block. The syntax for a variable declaration is shown below.

```
arg_type variables;
```

Here arg_type must be one of the types supported by SPC. Following the type are variable names, which must be a comma-separated list of identifiers with optional initial values as shown in the code fragment below.

```
name[=expression]
```

Arrays of variables may also be declared:

```
int array[n][=initializer];
```

You can also define variables using user-defined aggregate structure types.

```
struct TPerson {
  int age;
  string name;
};
TPerson bob; // cannot be initialized at declaration
```

3.3.2 Assignment

Once declared, variables may be assigned the value of an expression using the syntax shown in the code sample below.

```
variable assign_operator expression;
```

There are eleven different assignment operators. The most basic operator, '=', simply assigns the value of the expression to the variable. The other operators modify the variable's value in some other way as shown in the table below.

Operator	Action
=	Set variable to expression
+=	Add expression to variable
-=	Subtract expression from variable
*=	Multiple variable by expression
/=	Divide variable by expression

%=	Set variable to remainder after dividing by expression
&=	Bitwise AND expression into variable
|=	Bitwise OR expression into variable
^=	Bitwise exclusive OR into variable
>>=	Right shift variable by expression
<<=	Left shift variable by expression

Operators

The code sample below shows a few of the different types of operators that you can use in SPC expressions.

```
x = 2;  // set x to 2
y = 7;  // set y to 7
x += y; // x is 9, y is still 7
```

3.3.3 Control Structures

An SPC task or function usually contains a collection of nested control structures.

There are several types described below.

- The compound statement
- The if statement
- The if-else statement
- The while statement
- The do statement
- The for statement
- The repeat statement
- The switch statement
- The goto statement
- The until statement

3.3.3.1 The compound statement

The simplest control structure is a compound statement.

This is a list of statements enclosed within curly braces ('{' and '}'):

```
{
        x = 1;
        y = 2;
}
```

Although this may not seem very significant, it plays a crucial role in building more complicated control structures. Many control structures expect a single statement as their body. By using a compound statement, the same control structure can be used to control multiple statements.

3.3.3.2 The if statement

The if statement evaluates a condition.

If the condition is true, it executes one statement (the consequence). The value of a condition is considered to be false only when it evaluates to zero. If it evaluates to any non-zero value, it is true. The syntax for an if statement is shown below.

```
if (condition) consequence
```

The condition of an if-statement must be enclosed in parentheses, as shown in the code sample below. The compound statement in the last example allows two statements to execute as a consequence of the condition being true.

```
if (x==1) y = 2;
if (x==1) { y = 1; z = 2; }
```

3.3.3.3 The if-else statement

The if-else statement evaluates a condition.

If the condition is true, it executes one statement (the consequence). A second statement (the alternative), preceded by the keyword else, is executed if the condition is false. The value of a condition is considered to be false only when it evaluates to zero. If it evaluates to any non-zero value, it is true. The syntax for an if-else statement is shown below.

```
if (condition) consequence else alternative
```

The condition of an if-statement must be enclosed in parentheses, as shown in the code sample below. The compound statement in the last example allows two statements to execute as a consequence of the condition being true as well as two which execute when the condition is false.

```
if (x==1)
  y = 3;
else
  y = 4;
if (x==1) {
  y = 1;
  z = 2;
}
else {
  y = 3;
  z = 5;
}
```

3.3.3.4 The while statement

The while statement is used to construct a conditional loop.

The condition is evaluated, and if true the body of the loop is executed, then the condition is tested again. This process continues until the condition becomes false (or a break statement is executed). The syntax for a while loop appears in the code fragment below.

```
while (condition) body
```

Because the body of a while statement must be a single statement, it is very common to use a compound statement as the body. The sample below illustrates this usage pattern.

```
while(x < 10)
{
        x = x+1;
        y = y*2;
}
```

3.3.3.5 The do statement

A variant of the while loop is the do-while loop.

The syntax for this control structure is shown below.

```
do body while (condition)
```

The difference between a while loop and a do-while loop is that the do-while loop always executes the body at least once, whereas the while loop may not execute it at all.

```
do
{
        x = x+1;
        y = y*2;
} while(x < 10);
```

3.3.3.6 The for statement

Another kind of loop is the for loop.

This type of loop allows automatic initialization and incrmementation of a counter variable. It uses the syntax shown below.

```
for(statement1 ; condition ; statement2) body
```

A for loop always executes statement1, and then it repeatedly checks the condition. While the condition remains true, it executes the body followed by statement2. The for loop is equivalent to the code shown below.

```
statement1;
while(condition)
{
        body
        statement2;
}
```

Frequently, statement1 sets a loop counter variable to its starting value. The condition is generally a relational statement that checks the counter variable against a termination value, and statement2 increments or decrements the counter value.

Here is an example of how to use the for loop:

```
for (int i=0; i<8; i++)
{
         NumOut(0, LCD_LINE1-i*8, i);
}
```

3.3.3.7 The repeat statement

The repeat statement executes a loop a specified number of times.

This control structure is not included in the set of Standard C looping constructs. SPC inherits this statement from NQC. The syntax is shown below.

```
repeat (expression) body
```

The expression determines how many times the body will be executed. Note: the expression following the repeat keyword is evaluated a single time and then the body is repeated that number of times. This is different from both the while and do-while loops which evaluate their condition each time through the loop.

Here is an example of how to use the repeat loop:

```
int i=0;
repeat (8)
{
        printf("%d\n", i++);
}
```

3.3.3.8 The switch statement

A switch statement executes one of several different code sections depending on the value of an expression.

One or more case labels precede each code section. Each case must be a constant and unique within the switch statement. The switch statement evaluates the expression, and then looks for a matching case label. It will execute any statements following the matching case until either a break statement or the end of the switch is reached. A single default label may also be used - it will match any value not already appearing in a case label. A switch statement uses the syntax shown below.

```
switch (expression) body
```

Additional information about the case and default labels and the break statement can be found below.

- The case label
- The default label
- The break statement

A typical switch statement might look like this:

```
switch(x)
{
        case 1:
                // do something when x is 1
                break;
        case 2:
        case 3:
                // do something else when x is 2 or 3
                break;
        default:
                // do this when x is not 1, 2, or 3
                break;
}
```

3.3.3.8.1 The case label

The case label in a switch statement is not a statement in itself.

It is a label that precedes a list of statements. Multiple case labels can precede the same statement. The case label has the syntax shown below.

```
case constant_expression :
```

The switch statement page contains an example of how to use the case label.

3.3.3.8.2 The default label

The default label in a switch statement is not a statement in itself.

It is a label that precedes a list of statements. There can be only one default label within a switch statement. The default label has the syntax shown below.

```
default :
```

The switch statement page contains an example of how to use the default label.

3.3.3.9 The goto statement

The goto statement forces a program to jump to the specified location.

Statements in a program can be labeled by preceding them with an identifier and a colon. A goto statement then specifies the label that the program should jump to. You can only branch to a label within the current function or task, not from one function or task to another.

Here is an example of an infinite loop that increments a variable:

```
my_loop:
        x++;
        goto my_loop;
```

The goto statement should be used sparingly and cautiously. In almost every case, control structures such as if, while, and switch make a program much more readable and maintainable than using goto.

3.3.3.10 The until statement

SPC also defines an until macro for compatibility with NQC.

This construct provides a convenient alternative to the while loop. The actual definition of until is shown below.

```
#define until(c)        while(!(c))
```

In other words, until will continue looping until the condition becomes true. It is most often used in conjunction with an empty body statement or a body which simply yields to other tasks:

```
until(EVENT_OCCURS);    // wait for some event to occur
```

3.3.4 The asm statement

The asm statement is used to define many of the SPC API calls.

The syntax of the statement is shown below.

```
asm {
 one or more lines of SPRO assembly language
}
```

The statement simply emits the body of the statement as SuperPro ASM code and passes it directly to the compiler's backend. The asm statement can often be used to optimize code so that it executes as fast as possible on the SuperPro firmware. The following example shows an asm block containing variable declarations, labels, and basic SPRO ASM statements as well as comments.

```
asm {
      MVI WORK2, 12
      MOV PTR, WORK2
      MOV (PTR), WORK1
      INC PTR
}
```

The asm block statement and these special ASM keywords are used throughout the SPC API. You can have a look at the SPCDefs.h header file for several examples of how they are used. To keep the main SPC code as "C-like" as possible and for the sake of better readability SPC asm block statements can be wrapped in preprocessor macros and placed in custom header files which are included using #include.

3.3.5 Other SPC Statements

SPC supports a few other statement types.

The other SPC statements are described below.

- The function call statement
- The start statement
- The break statement
- The continue statement
- The return statement

Many expressions are not legal statements. A notable exception are expressions using increment (++) or decrement (–) operators.

```
x++;
```

The empty statement (just a bare semicolon) is also a legal statement.

3.3.5.1 The function call statement

A function call can also be a statement of the following form:

```
name(arguments);
```

The arguments list is a comma-separated list of expressions. The number and type of arguments supplied must match the definition of the function itself. Optionally, the return value may be assigned to a variable.

3.3.5.2 The start statement

You can start a task with the start statement.

This statement can be used with both the standard and enhanced NBC/SPC firmwares. The resulting operation is a native opcode in the enhanced firmware but it requires special compiler-generated subroutines in order to work with the standard firmware.

```
start task_name;
```

3.3.5.3 The break statement

Within loops (such as a while loop) you can use the break statement to exit the loop immediately.

It only exits out of the innermost loop

```
break;
```

The break statement is also a critical component of most switch statements. It prevents code in subsequent code sections from being executed, which is usually a programmer's intent, by immediately exiting the switch statement. Missing break statements in a switch are a frequent source of hard-to-find bugs.

Here is an example of how to use the break statement:

```
while (x<100) {
  x = get_new_x();
  if (button_pressed())
    break;
  process(x);
}
```

3.3.5.4 The continue statement

Within loops you can use the continue statement to skip to the top of the next iteration of the loop without executing any of the code in the loop that follows the continue statement.

```
continue;
```

Here is an example of how to use the continue statement:

```
while (x<100) {
  ch = get_char();
  if (ch != 's')
    continue;
  process(ch);
}
```

3.3.5.5 The return statement

If you want a function to return a value or to return before it reaches the end of its code, use a return statement.

An expression may optionally follow the return keyword and, when present, is the value returned by the function. The type of the expression must be compatible with the return type of the function.

```
return [expression];
```

3.4 Expressions

Values are the most primitive type of expressions.

More complicated expressions are formed from values using various operators.

Numerical constants in the SuperPro are represented as integer values. SPC internally uses 32 bit floating point math for constant expression evaluation. Numeric constants are written as either decimal (e.g. 123, 3.14) or hexadecimal (e.g. 0xABC). Presently, there is very little range checking on constants, so using a value larger than expected may produce unusual results.

Two special values are predefined: true and false. The value of false is zero (0), while the value of true is one (1). The same values hold for relational operators (e.g. $<$): when the relation is false the value is 0, otherwise the value is 1.

Values may be combined using operators. SPC operators are listed here in order of precedence from highest to lowest.

<table>
<tr><th>Operator</th><th>Description</th><th>Associativity</th><th>Restriction</th><th>Example</th></tr>
<tr><td>abs()</td><td>Absolute value</td><td>n/a</td><td></td><td>abs(x)</td></tr>
<tr><td>sign()</td><td>Sign of operand</td><td>n/a</td><td></td><td>sign(x)</td></tr>
<tr><td>++, –</td><td>Postfix incremen-t/decrement</td><td>left</td><td>variables only</td><td>x++</td></tr>
<tr><td>++, –</td><td>Prefix incremen-t/decrement</td><td>right</td><td>variables only</td><td>++x</td></tr>
<tr><td>-</td><td>Unary minus</td><td>right</td><td></td><td>-x</td></tr>
<tr><td>~</td><td>Bitwise negation (unary)</td><td>right</td><td></td><td>~123</td></tr>
<tr><td>!</td><td>Logical negation</td><td>right</td><td></td><td>!x</td></tr>
<tr><td>*, /, %</td><td>Multiplication, division, modulus</td><td>left</td><td></td><td>x * y</td></tr>
<tr><td>+, -</td><td>Addition, subtraction</td><td>left</td><td></td><td>x + y</td></tr>
<tr><td><<, >></td><td>Bitwise shift left and right</td><td>left</td><td></td><td>x << 4</td></tr>
<tr><td><, >, <=, >=</td><td>relational operators</td><td>left</td><td></td><td>x < y</td></tr>
<tr><td>==, !=</td><td>equal to, not equal to</td><td>left</td><td></td><td>x == 1</td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>left</td><td></td><td>x & y</td></tr>
<tr><td>^</td><td>Bitwise exclusive OR</td><td>left</td><td></td><td>x ^ y</td></tr>
<tr><td>|</td><td>Bitwise inclusive OR</td><td>left</td><td></td><td>x | y</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>left</td><td></td><td>x && y</td></tr>
<tr><td>||</td><td>Logical OR</td><td>left</td><td></td><td>x || y</td></tr>
<tr><td>?:</td><td>Ternary conditional value</td><td>right</td><td></td><td>x==1 ? y : z</td></tr>
</table>

Expression Operators

Where needed, parentheses are used to change the order of evaluation:

```
x = 2 + 3 * 4;  // set x to 14
y = (2 + 3) * 4;        // set y to 20
```

- Conditions

3.4.1 Conditions

Comparing two expressions forms a condition.

A condition may be negated with the logical negation operator, or two conditions combined with the logical AND and logical OR operators. Like most modern computer languages, SPC supports something called "short-circuit" evaluation of conditions. This means that if the entire value of the conditional can be logically determined by only evaluating the left hand term of the condition, then the right hand term will not be evaluated.

The table below summarizes the different types of conditions.

Condition	Meaning
Expr	true if expr is not equal to 0
Expr1 == expr2	true if expr1 equals expr2
Expr1 != expr2	true if expr1 is not equal to expr2
Expr1 $<$ expr2	true if one expr1 is less than expr2
Expr1 $<$= expr2	true if expr1 is less than or equal to expr2
Expr1 $>$ expr2	true if expr1 is greater than expr2
Expr1 $>$= expr2	true if expr1 is greater than or equal to expr2
! condition	logical negation of a condition - true if condition is false
Cond1 && cond2	logical AND of two conditions (true if and only if both conditions are true)
Cond1 $\|\|$ cond2	logical OR of two conditions (true if and only if at least one of the conditions are true)

Conditions

There are also two special constant conditions which can be used anywhere that the above conditions are allowed. They are listed below.

- The true condition
- The false condition

You can use conditions in SPC control structures, such as the if-statement and the while or until statements, to specify exactly how you want your program to behave.

3.4.1.1 The true condition

The keyword true has a value of one.

It represents a condition that is always true.

3.4.1.2 The false condition

The keyword false has a value of zero.

It represents a condition that is always false.

3.5 The Preprocessor

SPC also includes a preprocessor that is modeled after the Standard C preprocessor.

The C preprocessor processes a source code file before the compiler does. It handles such tasks as including code from other files, conditionally including or excluding blocks of code, stripping comments, defining simple and parameterized macros, and expanding macros wherever they are encountered in the source code.

The SPC preprocessor implements the following standard preprocessor directives: #include, #define, #ifdef, #ifndef, #endif, #if, #elif, #undef, ##, #line, #error, and #pragma. Its implementation is close to a standard C preprocessor's, so most preprocessor directives should work as C programmers expect in SPC. Any significant deviations are explained below.

- #include
- #define
- ## (Concatenation)

- Conditional Compilation
- Pragmas

3.5.1 #include

The #include command works as in Standard C, with the caveat that the filename must be enclosed in double quotes.

There is no notion of a system include path, so enclosing a filename in angle brackets is forbidden.

```
#include "foo.h"  // ok
#include <foo.h> // error!
```

SPC programs can begin with #include "NXCDefs.h" but they don't need to. This standard header file includes many important constants and macros, which form the core SPC API. SPC no longer require that you manually include the NXCDefs.h header file. Unless you specifically tell the compiler to ignore the standard system files, this header file is included automatically.

3.5.2 #define

The #define command is used for macro substitution.

Redefinition of a macro will result in a compiler warning. Macros are normally restricted to one line because the newline character at the end of the line acts as a terminator. However, you can write multiline macros by instructing the preprocessor to ignore the newline character. This is accomplished by escaping the newline character with a backslash ('\'). The backslash character must be the very last character in the line or it will not extend the macro definition to the next line. The code sample below shows how to write a multi-line preprocessor macro.

```
#define foo(x)  do { bar(x); \\
                     baz(x); } while(false)
```

The #undef directive may be used to remove a macro's definition.

3.5.3 ## (Concatenation)

The ## directive works similar to the C preprocessor.

It is replaced by nothing, which causes tokens on either side to be concatenated together. Because it acts as a separator initially, it can be used within macro functions to produce identifiers via combination with parameter values.

3.5.4 Conditional Compilation

Conditional compilation works similar to the C preprocessor's conditional compilation.

The following preprocessor directives may be used:

Directive	Meaning
#ifdef symbol	If symbol is defined then compile the following code
#ifndef symbol	If symbol is not defined then compile the following code
#else	Switch from compiling to not compiling and vice versa
#endif	Return to previous compiling state
#if condition	If the condition evaluates to true then compile the following code
#elif	Same as #else but used with #if

Conditional compilation directives

See the SPCDefs.h header files for many examples of how to use conditional compilation.

3.5.5 Pragmas

The '#pragma' directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself.

A C compiler is free to attach any meaning it likes to pragmas.

In SuperPro C the only pragma that has any significant meaning is 'autostart'.

```
#pragma autostart
```

The autostart instruction tells the compiler to modify the generated executable so that it automatically starts running whenever the SuperPro is powered on.

4 Pre-defined system constants

Pre-defined system constants for directly interacting with the SuperPro hardware.

The spmem.h header file uses system constants to define names for the I/O mapped memory addresses of the SuperPro board you are targetting. A complete list of these system constants appears below:

- ADChannel0/1/2/3
- DigitalIn
- DigitalOut
- DigitalControl
- StrobeControl
- Timer0/1/2/3
- SerialInCount
- SerialInByte
- SerialOutCount
- SerialOutByte
- DAC0Mode/DAC1Mode
- DAC0Frequency/DAC1Frequency
- DAC0Voltage/DAC1Voltage
- LEDControl
- SystemClock
- Shared Memory

4.1 ADChannel0/1/2/3

These variables return the voltage on pins A0/1/2/3 as a value in the range 0 - 1023.

This range of values represents a voltage range of 0 - 3.3 volts, or ∼3.222 mV per step.

```
//convert channel 0 reading to millivolts
int voltage = ( ADChannel0 * 3222 ) / 1000 ;
```

4.2 DigitalIn

This variable returns the current state of the 8 digital lines, B0 - B7.

This includes the state of any of the lines which are configured as outputs.

```
if ( DigitalIn & DIGI_PIN7 == DIGI_PIN7 )  //check
       if bit 7 set
{
  // do something here
}
```

4.3 DigitalOut

This variable sets the current state of any of the 8 digital lines, B0 - B7 which are set as outputs.

See Digital pin constants.

```
DigitalOut = DIGI_PIN0 ;   //set B0
```

4.4 DigitalControl

This variable defines which of the 8 digital lines, B0 - B7, are set as outputs.

If the corresponding bit in 1, the line is configured as an output, else it will be an input. See Digital pin constants.

```
DigitalControl = 0x0F ;   //set DIGI_PIN0 - DIGI_PIN3 as outputs
```

4.5 StrobeControl

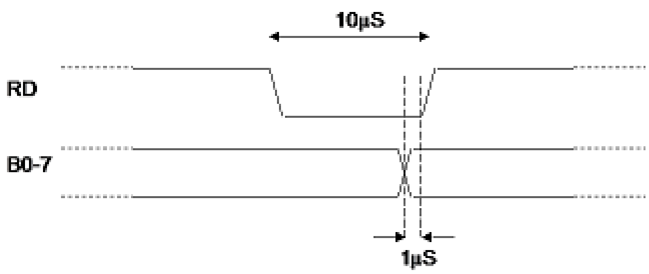
This variable allows control over the 6 strobe lines.

See Strobe control constants.

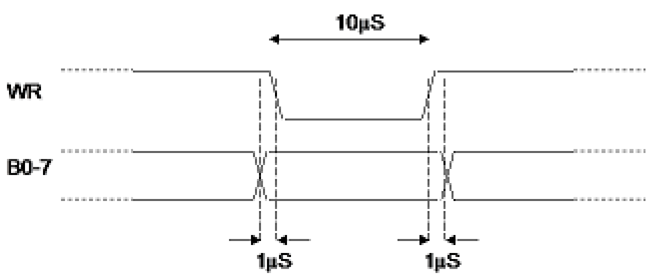
D31-D6	D5	D4	D3	D2	D1	D0
-	WR	RD	S3	S2	S1	S0

Strobe Lines

There are 4 general purpose outputs, S0 - S3. These 4 lines may be used as digital outputs. There are 2 special purpose outputs, RD and WR. These lines are automatically activated when DigitalIn is read or DigitalOut is written. When DigitalIn is read, the RD output will pulse for about 10 S. If the StrobeControl RD bit is 0, the RD output will pulse high, if the bit is 1, the output will pulse low.

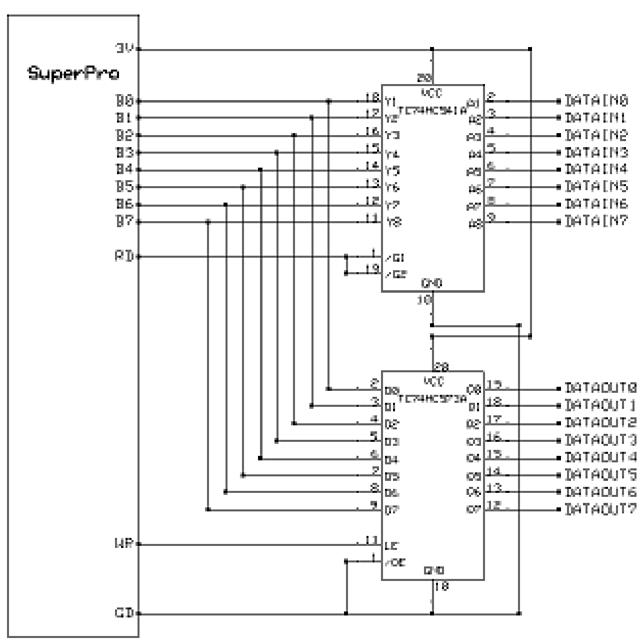


The timing for a read. An external device which is relying on the RD strobe output for synchronizing with the SuperPro hardware may use the leading edge of the RD strobe to present data on B0 - B7. The device must have the data ready within 9 microseconds of the start (leading edge) of the RD strobe. The timing for a write.



An external device which is relying on the WR strobe output for synchronizing with the SuperPro hardware may use the leading edge of the WR strobe as either an edge type clock or a latch type due to the data being presented on B0 - B7 at least 1 microsecond before the strobe is active until 1 microsecond after.

```
DigitalControl = 0xFF ;    //set B0 - B7 as outputs
DigitalOut = outputbyte ;
DigitalControl = 0x00 ;    //reset B0 - B7 to inputs
```



In a typical example of using the strobes to use the B0-7 bus bi-directionally:

```
StrobeControl = 0x10 ; //set RD active low and WR active high
DigitalControl = 0x00 ;    //ensure outputs are inactive
```

```
// ...
datain = DigitalIn ; //read DATAIN byte
// ...
DigitalControl = 0xFF ;   //set B0 - B7 as outputs
DigitalOut = dataout ;    //write DATAOUT byte
DigitalControl = 0x00 ;   //reset B0 - B7 to inputs
```

4.6 Timer0/1/2/3

The timers are count-down and halt at zero types.

They count down at the rate of 1000 counts per second, i.e., one count per millisecond.

```
Timer1 = 1000 ;  //set timer1 to run for 1 second
while ( Timer1 != 0 ) ;   /wait for timer1 to expire
```

4.7 SerialInCount

The SerialInCount returns the number of characters waiting in an input FIFO (First In, First Out) buffer of up to 255 entries.

Characters can be transferred from the host PC to the SuperPro via a terminal emulation program running at 115200bps, 8 bits, no parity. A program can wait for a character to become available using this value.

```
while ( SerialInCount == 0 ) ; //wait for a character
```

4.8 SerialInByte

The SerialInByte returns the character waiting in the input FIFO receive queue.

A program should wait for a character to become available before performing a read from SerialInByte. The result of reading from an empty FIFO receive queue is unpredictable.

```
while ( SerialInCount == 0 ) ; //wait for a character
kbchar = SerialInByte ;  //get it
```

4.9 SerialOutCount

The SerialOutCount returns the number of characters waiting in an output FIFO send queue of 255 entries.

Characters from the output FIFO are transferred to the host PC at approximately 10,000 per second. If the program is generating characters at a rate greater than this, the output FIFO will start to fill up. This state can be checked by the program by comparing the count with SERIAL_BUFFER_SIZE. In the event that this check is not performed, no data will be lost since the program will stall waiting for space to become available in the FIFO.

```
while ( SerialOutCount > 254 ) ; //wait for space for a character
```

4.10 SerialOutByte

The SerialOutByte sends a character to the output FIFO send queue.

The result of writing to a full FIFO send queue is to cause the program to stall. The send queue can hold up to 255 bytes.

```
while ( SerialOutCount > 254 ) ; //wait for space for a character
SerialOutByte = 'C' ;  //send a byte
```

4.11 DAC0Mode/DAC1Mode

The DACnMode controls the operation of the analog output pins O0/O1.

The following modes are available for use:

Mode	Value	Function
DAC_MODE_DCOUT	0	Steady (DC) voltage output
DAC_MODE_SINEWAVE	1	Sine wave output
DAC_MODE_SQUAREWAVE	2	Square wave output
DAC_MODE_SAWPOSWAVE	3	Positive going sawtooth output
DAC_MODE_SAWNEGWAVE	4	Negative going sawtooth output
DAC_MODE_TRIANGLEWAVE	5	Triangle wave output
DAC_MODE_PWMVOLTAGE	6	PWM square wave output
DAC_MODE_RESTART_MASK	0x80	Restart waveform mask

Analog Output Modes

Mode DAC_MODE_DCOUT uses the DACnVoltage to control the output voltage between 0 and 3.3 volts in steps of 3.222 mV.

The waveforms associated with modes DAC_MODE_SINEWAVE, DAC_MODE_SQUAREWAVE, DAC_MODE_SAW-POSWAVE, DAC_MODE_SAWNEGWAVE, and DAC_MODE_TRIANGLEWAVE are centered around a DC offset of 1.65 volts. The DACnVoltage controls the amplitude from +/- 0 to +/- 1.65 volts.

The waveform associated with mode DAC_MODE_PWMVOLTAGE is a rectangular waveform switching between 0 and 3.3 volts. The DACnVoltage controls the mark to space ratio between 0% and 100%. The average DC value of this waveform thus varies from 0 to 3.3 volts.

```
DAC0Mode = DAC_MODE_PWMVOLTAGE; // use PWM output
DAC0Frequency = 4000; // 4khz frequency.
DAC0Voltage = 512; // mark/space ratio = 50%
```

As of firmware version 2.3 the waveform generator has been improved to provide:

- Seamless change in output frequency if the generator is already running.
- Start of waveform generation will start from the beginning of the wave table. When the DAC0/DAC1 Mode bytes are changed from 0x00 to a non zero number, the waveform generation will start from the start of the wave table.
- A frequency value of zero is interpreted as a DC output of 1.65v, the center offset value used for waveform output.
- Addition of bit 7 of the DAC0/DAC1 Mode bytes to signify force waveform generation from the beginning of the wave table. If the Mode byte is written with bit 7 set, i.e., 0x81 for a sine wave, then the wave table pointer will reset to the start of the table. Once this reset has been performed, the 7 bit will be cleared.
- Immediate cessation of waveform generation when the DAC0/DAC1 Mode bytes are set to 0x00. The previous firmware version would inadvertently insert a 0 - 4mS dead time when the DAC0/DAC1 Mode bytes were changed to 0x00. The reaction is now a few microseconds instead.

4.12 DAC0Frequency/DAC1Frequency

The DACnFrequency controls the generator frequency for the analog output pins O0/O1 for DACnModes DAC_MODE-_SINEWAVE, DAC_MODE_SQUAREWAVE, DAC_MODE_SAWPOSWAVE, DAC_MODE_SAWNEGWAVE, DAC_M-ODE_TRIANGLEWAVE, and DAC_MODE_PWMVOLTAGE.

The available frequency range is 1 - 8000 Hz.

```
DAC0Mode = DAC_MODE_SINEWAVE; // use sine wave output
DAC0Frequency = 4000; // 4khz frequency.
DAC0Voltage = 1024; // full range amplitude
```

4.13 DAC0Voltage/DAC1Voltage

The DACnVoltage controls the output voltage levels for the analog output pins O0/O1.

DACnMode DAC_MODE_DCOUT uses the DACnVoltage to control the output voltage between 0 and 3.3 volts in steps of 3.222 mV.

For DACnModes DAC_MODE_SINEWAVE, DAC_MODE_SQUAREWAVE, DAC_MODE_SAWPOSWAVE, DAC_MODE_SAWNEGWAVE, and DAC_MODE_TRIANGLEWAVE, the DACnVoltage controls the amplitude from +/- 0 to +/- 1.65 volts.

For DACnMode DAC_MODE_PWMVOLTAGE, DACnVoltage controls the mark to space ratio between 0% and 100%. The average DC value of this waveform thus varies from 0 to 3.3 volts.

```
DAC0Mode = DAC_MODE_DCOUT; // DC ouput voltage
DAC0Voltage = 500; // set voltage level of O0 to 500*3.222 mV
```

4.14 LEDControl

The LEDControl location can be used to turn two on-board LEDs on and off.

Bit 0 controls the state of a red LED, while bit 1 controls a blue LED.

```
LEDControl = LED_BLUE | LED_RED; // turn on both the
       blue and red LEDs
```

4.15 SystemClock

The SystemClock returns the number of milliseconds since power was applied to the SuperPro board.

```
long x = SystemClock;
```

4.16 Shared Memory

There are 32 shared memory locations on the SuperPro board which can be accessed remotely via I2C using a device like the NXT.

They are named SharedMem01 through SharedMem32. If you write 4 bytes to I2C address 0x80 these bytes can be read as a signed long value in SuperPro C at address 0x20 (SharedMem01).

```
long x = SharedMem01;
SharedMem32 = ADChannel0;
```

5 Module Documentation

5.1 Miscellaneous constants

Miscellaneous constants for use in SPC.

Modules

- Data type limits
  Constants that define various data type limits.

Macros

- #define TRUE 1
- #define FALSE 0
- #define SERIAL_BUFFER_SIZE 255

5.1.1 Detailed Description

Miscellaneous constants for use in SPC.

5.1.2 Macro Definition Documentation

5.1.2.1 #define FALSE 0

A false value

5.1.2.2 #define SERIAL_BUFFER_SIZE 255

Serial port receive and send buffer size

5.1.2.3 #define TRUE 1

A true value

5.2 Pre-defined System constants

Pre-defined system constants for use in SuperPro C to read or read/write all of the SuperPro hardware capabilities.

Macros

- #define ADChannel0 @0x00
- #define ADChannel1 @0x01
- #define ADChannel2 @0x02
- #define ADChannel3 @0x03
- #define DigitalIn @0x08
- #define DigitalOut @0x09
- #define DigitalControl @0x0A
- #define StrobeControl @0x0B
- #define Timer0 @0x0C
- #define Timer1 @0x0D
- #define Timer2 @0x0E
- #define Timer3 @0x0F
- #define SerialInCount @0x10
- #define SerialInByte @0x11
- #define SerialOutCount @0x12
- #define SerialOutByte @0x13
- #define DAC0Mode @0x18
- #define DAC0Frequency @0x19
- #define DAC0Voltage @0x1A
- #define DAC1Mode @0x1B
- #define DAC1Frequency @0x1C
- #define DAC1Voltage @0x1D
- #define LEDControl @0x1E
- #define SystemClock @0x1F

5.2.1 Detailed Description

Pre-defined system constants for use in SuperPro C to read or read/write all of the SuperPro hardware capabilities.

5.2.2 Macro Definition Documentation

5.2.2.1 #define ADChannel0 @0x00

Reads the current voltage on A0 input. Value ranges from 0 to 1023. Updated every millisecond. Read only.

Examples:

ex_abort.spc, ex_arrays.spc, ex_digiserial.spc, ex_ledtest.spc, ex_printf.spc, and ex_rotate.spc.

5.2.2.2 #define ADChannel1 @0x01

Reads the current voltage on A1 input. Value ranges from 0 to 1023. Updated every millisecond. Read only.

Examples:

ex_digiserial.spc, and ex_ledtest.spc.

5.2.2.3 #define ADChannel2 @0x02

Reads the current voltage on A2 input. Value ranges from 0 to 1023. Updated every millisecond. Read only.

5.2.2.4 #define ADChannel3 @0x03

Reads the current voltage on A3 input. Value ranges from 0 to 1023. Updated every millisecond. Read only.

Examples:

ex_DAC0ToA3.spc.

5.2.2.5 #define DAC0Frequency @0x19

Control the frequency of the DAC0 analog output (O0). Read/write.

Examples:

ex_DAC0ToA3.spc.

5.2.2.6 #define DAC0Mode @0x18

Control the operation of the DAC0 analog output (O0). See Analog output mode constants for valid values. Read-/write.

Examples:

ex_DAC0ToA3.spc.

5.2.2.7 #define DAC0Voltage @0x1A

Control the voltage of the DAC0 analog output (O0). Read/write.

Examples:

ex_DAC0ToA3.spc.

5.2.2.8 #define DAC1Frequency @0x1C

Control the frequency of the DAC1 analog output (O1). Read/write.

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

5.2.2.9 #define DAC1Mode @0x1B

Control the operation of the DAC1 analog output (O1). See Analog output mode constants for valid values. Read-/write.

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

5.2.2.10 #define DAC1Voltage @0x1D

Control the voltage of the DAC1 analog output (O1). Read/write.

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

5.2.2.11 #define DigitalControl @0x0A

Write 8 bits to the digital control port B0 - B7. Set the mode of any of the 8 digital signals. 1 == output, 0 == input.

Examples:

ex_digiserial.spc, and ex_ledtest.spc.

5.2.2.12 #define DigitalIn @0x08

Read 8 bits from the digital port B0 - B7. Read only.

5.2.2.13 #define DigitalOut @0x09

Write 8 bits to the digital port B0 - B7. Read/Write.

Examples:

ex_digiserial.spc, and ex_ledtest.spc.

5.2.2.14 #define LEDControl @0x1E

Control the operation of the two onboard LDEs (red and blue). See LED control constants for valid values. Read-/write.

Examples:

ex_ledcontrol.spc.

5.2.2.15 #define SerialInByte @0x11

Read the next serial byte from the serial port receive queue. Reading this value removes the byte from the receive queue. Serial port input data is stored in a 255 byte temporary buffer. Read only.

Examples:

ex_digiserial.spc.

5.2.2.16 #define SerialInCount @0x10

Read the count of serial bytes in the receive queue. Enables a user program to check if any data is available to be read from the serial port. Read only.

Examples:

ex_digiserial.spc.

5.2.2.17 #define SerialOutByte @0x13

Write a byte to the serial port send queue. Serial port output data is stored in a 255 byte temporary buffer. Do not write to this address if SerialCount is 255. Write only.

Examples:

ex_serialout.spc.

5.2.2.18 #define SerialOutCount @0x12

Read the count of serial bytes in the send queue. Enables a user program to check how many bytes are waiting to be sent out the serial port. Read only.

Examples:

ex_serialout.spc.

5.2.2.19 #define StrobeControl @0x0B

Write 6 bits to the digital strobe port S0 - WR. Controls the operation of the six strobe outputs (S0, S1, S2, S3, RD, and WR). See Strobe control constants for valid values.

5.2.2.20 #define SystemClock @0x1F

Read the system clock. The system clock counts up continuously at one count per millisecond. Read only.

Examples:

ex_DAC0ToA3.spc, ex_systemclock.spc, ex_wait.spc, and ex_yield.spc.

5.2.2.21 #define Timer0 @0x0C

Read/write countdown timer 0. Counts down until it reaches zero (per millisecond).

Examples:

ex_abs.spc, ex_digiserial.spc, ex_ledtest.spc, ex_stat.spc, ex_Stop.spc, and ex_timer.spc.

5.2.2.22 #define Timer1 @0x0D

Read/write countdown timer 1. Counts down until it reaches zero (per millisecond).

Examples:

ex_timer.spc, and ex_write.spc.

5.2.2.23 #define Timer2 @0x0E

Read/write countdown timer 2. Counts down until it reaches zero (per millisecond).

Examples:

ex_timer.spc.

5.2.2.24 #define Timer3 @0x0F

Read/write countdown timer 3. Counts down until it reaches zero (per millisecond).

Examples:

ex_pushpop.spc, and ex_timer.spc.

5.3 Pre-defined shared memory constants

Pre-defined shared memory constants for use in SuperPro C to read/write addresses in SuperPro memory that are shared with the I2C address space.

Macros

- #define SharedMem01 @0x20
- #define SharedMem02 @0x21
- #define SharedMem03 @0x22
- #define SharedMem04 @0x23
- #define SharedMem05 @0x24
- #define SharedMem06 @0x25
- #define SharedMem07 @0x26
- #define SharedMem08 @0x27
- #define SharedMem09 @0x28
- #define SharedMem10 @0x29
- #define SharedMem11 @0x2a
- #define SharedMem12 @0x2b
- #define SharedMem13 @0x2c
- #define SharedMem14 @0x2d
- #define SharedMem15 @0x2e
- #define SharedMem16 @0x2f
- #define SharedMem17 @0x30
- #define SharedMem18 @0x31
- #define SharedMem19 @0x32
- #define SharedMem20 @0x33
- #define SharedMem21 @0x34
- #define SharedMem22 @0x35
- #define SharedMem23 @0x36
- #define SharedMem24 @0x37
- #define SharedMem25 @0x38
- #define SharedMem26 @0x39
- #define SharedMem27 @0x3a
- #define SharedMem28 @0x3b
- #define SharedMem29 @0x3c
- #define SharedMem30 @0x3d
- #define SharedMem31 @0x3e
- #define SharedMem32 @0x3f

5.3.1 Detailed Description

Pre-defined shared memory constants for use in SuperPro C to read/write addresses in SuperPro memory that are shared with the I2C address space. This allows for an external device to send data to or receive data from a SuperPro program while it is executing using the I2C communication layer.

5.3.2 Macro Definition Documentation

5.3.2.1 #define SharedMem01 @0x20

Read/write via I2C at address 0x80 (msb/lsb)

5.3.2.2 #define SharedMem02 @0x21

Read/write via I2C at address 0x84 (msb/lsb)

5.3.2.3 #define SharedMem03 @0x22

Read/write via I2C at address 0x88 (msb/lsb)

5.3.2.4 #define SharedMem04 @0x23

Read/write via I2C at address 0x8C (msb/lsb)

5.3.2.5 #define SharedMem05 @0x24

Read/write via I2C at address 0x90 (msb/lsb)

5.3.2.6 #define SharedMem06 @0x25

Read/write via I2C at address 0x94 (msb/lsb)

5.3.2.7 #define SharedMem07 @0x26

Read/write via I2C at address 0x98 (msb/lsb)

5.3.2.8 #define SharedMem08 @0x27

Read/write via I2C at address 0x9C (msb/lsb)

5.3.2.9 #define SharedMem09 @0x28

Read/write via I2C at address 0xA0 (msb/lsb)

5.3.2.10 #define SharedMem10 @0x29

Read/write via I2C at address 0xA4 (msb/lsb)

5.3.2.11 #define SharedMem11 @0x2a

Read/write via I2C at address 0xA8 (msb/lsb)

5.3.2.12 #define SharedMem12 @0x2b

Read/write via I2C at address 0xAC (msb/lsb)

5.3.2.13 #define SharedMem13 @0x2c

Read/write via I2C at address 0xB0 (msb/lsb)

5.3.2.14 #define SharedMem14 @0x2d

Read/write via I2C at address 0xB4 (msb/lsb)

5.3.2.15 #define SharedMem15 @0x2e

Read/write via I2C at address 0xB8 (msb/lsb)

5.3.2.16 #define SharedMem16 @0x2f

Read/write via I2C at address 0xBC (msb/lsb)

5.3.2.17 #define SharedMem17 @0x30

Read/write via I2C at address 0xC0 (msb/lsb)

5.3.2.18 #define SharedMem18 @0x31

Read/write via I2C at address 0xC4 (msb/lsb)

5.3.2.19 #define SharedMem19 @0x32

Read/write via I2C at address 0xC8 (msb/lsb)

5.3.2.20 #define SharedMem20 @0x33

Read/write via I2C at address 0xCC (msb/lsb)

5.3.2.21 #define SharedMem21 @0x34

Read/write via I2C at address 0xD0 (msb/lsb)

5.3.2.22 #define SharedMem22 @0x35

Read/write via I2C at address 0xD4 (msb/lsb)

5.3.2.23 #define SharedMem23 @0x36

Read/write via I2C at address 0xD8 (msb/lsb)

5.3.2.24 #define SharedMem24 @0x37

Read/write via I2C at address 0xDC (msb/lsb)

5.3.2.25 #define SharedMem25 @0x38

Read/write via I2C at address 0xE0 (msb/lsb)

5.3.2.26 #define SharedMem26 @0x39

Read/write via I2C at address 0xE4 (msb/lsb)

5.3.2.27 #define SharedMem27 @0x3a

Read/write via I2C at address 0xE8 (msb/lsb)

5.3.2.28 #define SharedMem28 @0x3b

Read/write via I2C at address 0xEC (msb/lsb)

5.3.2.29 #define SharedMem29 @0x3c

Read/write via I2C at address 0xF0 (msb/lsb)

5.3.2.30 #define SharedMem30 @0x3d

Read/write via I2C at address 0xF4 (msb/lsb)

5.3.2.31 #define SharedMem31 @0x3e

Read/write via I2C at address 0xF8 (msb/lsb)

5.3.2.32 #define SharedMem32 @0x3f

Read/write via I2C at address 0xFC (msb/lsb)

5.4 Analog output mode constants

Constants for controlling the 2 analog output modes.

Macros

- #define DAC_MODE_DCOUT 0
- #define DAC_MODE_SINEWAVE 1
- #define DAC_MODE_SQUAREWAVE 2
- #define DAC_MODE_SAWPOSWAVE 3
- #define DAC_MODE_SAWNEGWAVE 4
- #define DAC_MODE_TRIANGLEWAVE 5
- #define DAC_MODE_PWMVOLTAGE 6
- #define DAC_MODE_RESTART_MASK 0x80

5.4.1 Detailed Description

Constants for controlling the 2 analog output modes. Two analog outputs, which can span 0 to 3.3 volts, can be programmed to output a steady voltage or can be programmed to output a selection of waveforms over a range of frequencies.

In the DC output mode, the DAC0/DAC1 voltage fields control the voltage on the two analog outputs in increments of ~3.2mV from 0 - 1023 giving 0 - 3.3v.

In waveform modes, the channel outputs will center on 1.65 volts when generating waveforms. The DAC0/DAC1 voltage fields control the signal levels of the waveforms by adjusting the peak to peak signal levels from 0 - 3.3v.

In PWFM voltage mode, the channel outputs will create a variable mark:space ratio square wave at 3.3v signal level. The average output voltage is set by the O0/O1 voltage fields.

See Also

DAC0Mode, DAC1Mode

5.4.2 Macro Definition Documentation

5.4.2.1 #define DAC_MODE_DCOUT 0

Steady (DC) voltage output.

5.4.2.2 #define DAC_MODE_PWMVOLTAGE 6

PWM square wave output.

5.4.2.3 #define DAC_MODE_RESTART_MASK 0x80

Add mask to DAC mode constants to force waveform generation from the start of the wave table.

5.4.2.4 #define DAC_MODE_SAWNEGWAVE 4

Negative going sawtooth output.

5.4.2.5 #define DAC_MODE_SAWPOSWAVE 3

Positive going sawtooth output.

5.4.2.6 #define DAC_MODE_SINEWAVE 1

Sine wave output.

Examples:

ex_DAC0ToA3.spc.

5.4.2.7 #define DAC_MODE_SQUAREWAVE 2

Square wave output.

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

5.4.2.8 #define DAC_MODE_TRIANGLEWAVE 5

Triangle wave output.

5.5 LED control constants

Constants for controlling the 2 onboard LEDs.

Macros

- #define LED_BLUE 0x02
- #define LED_RED 0x01

5.5.1 Detailed Description

Constants for controlling the 2 onboard LEDs.

See Also

LEDControl

5.5.2 Macro Definition Documentation

5.5.2.1 #define LED_BLUE 0x02

Turn on the blue onboard LED.

Examples:

ex_ledcontrol.spc.

5.5.2.2 #define LED_RED 0x01

Turn on the red onboard LED.

Examples:

ex_ledcontrol.spc.

5.6 Digital pin constants

Constants for controlling the 8 digital pins.

Macros

- #define DIGI_PIN0 0x01
- #define DIGI_PIN1 0x02
- #define DIGI_PIN2 0x04
- #define DIGI_PIN3 0x08
- #define DIGI_PIN4 0x10
- #define DIGI_PIN5 0x20
- #define DIGI_PIN6 0x40
- #define DIGI_PIN7 0x80

5.6.1 Detailed Description

Constants for controlling the 8 digital pins. The eight digital inputs are returned as a byte representing the state of the eight inputs. The eight digital outputs are controlled by two bytes, the first of which sets the state of any of the signals which have been defined as outputs and the second of which controls the input/output state of each signal.

See Also

DigitalControl

5.6.2 Macro Definition Documentation

5.6.2.1 #define DIGI_PIN0 0x01

Access digital pin 0 (B0)

5.6.2.2 #define DIGI_PIN1 0x02

Access digital pin 1 (B1)

5.6.2.3 #define DIGI_PIN2 0x04

Access digital pin 2 (B2)

5.6.2.4 #define DIGI_PIN3 0x08

Access digital pin 3 (B3)

5.6.2.5 #define DIGI_PIN4 0x10

Access digital pin 4 (B4)

5.6.2.6 #define DIGI_PIN5 0x20

Access digital pin 5 (B5)

5.6.2.7 #define DIGI_PIN6 0x40

Access digital pin 6 (B6)

5.6.2.8 #define DIGI_PIN7 0x80

Access digital pin 7 (B7)

5.7 Strobe control constants

Constants for manipulating the six digital strobe outputs.

Macros

- #define STROBE_S0 0x01
- #define STROBE_S1 0x02
- #define STROBE_S2 0x04
- #define STROBE_S3 0x08
- #define STROBE_READ 0x10
- #define STROBE_WRITE 0x20

5.7.1 Detailed Description

Constants for manipulating the six digital strobe outputs. Six digital strobe outputs are available. One is pre-configured as a read strobe, another is pre-configured as a write strobe while the other four can be set to a high or low logic level. These strobe lines enable external devices to synchronize with the digital data port and multiplex the eight digital input/output bits to wider bit widths.

The RD and WR bits set the inactive state of the read and write strobe outputs. Thus, if these bits are set to 0, the strobe outputs will pulse high.

See Also

StrobeControl

5.7.2 Macro Definition Documentation

5.7.2.1 #define STROBE_READ 0x10

Access read pin (RD)

5.7.2.2 #define STROBE_S0 0x01

Access strobe 0 pin (S0)

5.7.2.3 #define STROBE_S1 0x02

Access strobe 1 pin (S1)

5.7.2.4 #define STROBE_S2 0x04

Access strobe 2 pin (S2)

5.7.2.5 #define STROBE_S3 0x08

Access strobe 3 pin (S3)

5.7.2.6 #define STROBE_WRITE 0x20

Access write pin (WR)

5.8 Data type limits

Constants that define various data type limits.

Macros

- #define CHAR_BIT 32
- #define LONG_MIN -2147483648
- #define SCHAR_MIN -2147483648
- #define INT_MIN -2147483648
- #define CHAR_MIN -2147483648
- #define LONG_MAX 2147483647
- #define SCHAR_MAX 2147483647
- #define INT_MAX 2147483647
- #define CHAR_MAX 2147483647

5.8.1 Detailed Description

Constants that define various data type limits.

5.8.2 Macro Definition Documentation

5.8.2.1 #define CHAR_BIT 32

The number of bits in the char type

5.8.2.2 #define CHAR_MAX 2147483647

The maximum value of the char type

5.8.2.3 #define CHAR_MIN -2147483648

The minimum value of the char type

5.8.2.4 #define INT_MAX 2147483647

The maximum value of the int type

5.8.2.5 #define INT_MIN -2147483648

The minimum value of the int type

5.8.2.6 #define LONG_MAX 2147483647

The maximum value of the long type

5.8.2.7 #define LONG_MIN -2147483648

The minimum value of the long type

5.8.2.8 #define SCHAR_MAX 2147483647

The maximum value of the signed char type

5.8.2.9 #define SCHAR_MIN -2147483648

The minimum value of the signed char type

5.9 Program slot constants

Constants for use with the Run() function.

Macros

- #define SLOT1 0
- #define SLOT2 1
- #define SLOT3 2
- #define SLOT4 3
- #define SLOT5 4
- #define SLOT6 5
- #define SLOT7 6

5.9.1 Detailed Description

Constants for use with the Run() function.

See Also

Run()

5.9.2 Macro Definition Documentation

5.9.2.1 #define SLOT1 0

Program slot 1.

5.9.2.2 #define SLOT2 1

Program slot 2.

Examples:

ex_run.spc.

5.9.2.3 #define SLOT3 2

Program slot 3.

5.9.2.4 #define SLOT4 3

Program slot 4.

5.9.2.5 #define SLOT5 4

Program slot 5.

5.9.2.6 #define SLOT6 5

Program slot 6.

5.9.2.7 #define SLOT7 6

Program slot 7.

5.10 Log status constants

Constants for use with the stat() function.

Macros

- #define LOG_STATUS_OPEN 2
- #define LOG_STATUS_BUSY 1
- #define LOG_STATUS_CLOSED 0

5.10.1 Detailed Description

Constants for use with the stat() function.

See Also

stat()

5.10.2 Macro Definition Documentation

5.10.2.1 #define LOG_STATUS_BUSY 1

Log file is busy.

5.10.2.2 #define LOG_STATUS_CLOSED 0

Log file is closed.

5.10.2.3 #define LOG_STATUS_OPEN 2

Log file is open.

Examples:

ex_stat.spc.

5.11 Time constants

Constants for use with the Wait() function.

Macros

- #define MS_1 1
- #define MS_2 2
- #define MS_3 3
- #define MS_4 4
- #define MS_5 5
- #define MS_6 6
- #define MS_7 7
- #define MS_8 8
- #define MS_9 9
- #define MS_10 10
- #define MS_20 20
- #define MS_30 30
- #define MS_40 40
- #define MS_50 50
- #define MS_60 60
- #define MS_70 70
- #define MS_80 80
- #define MS_90 90
- #define MS_100 100
- #define MS_150 150
- #define MS_200 200
- #define MS_250 250
- #define MS_300 300
- #define MS_350 350
- #define MS_400 400
- #define MS_450 450
- #define MS_500 500
- #define MS_600 600
- #define MS_700 700
- #define MS_800 800
- #define MS_900 900
- #define SEC_1 1000
- #define SEC_2 2000
- #define SEC_3 3000
- #define SEC_4 4000
- #define SEC_5 5000
- #define SEC_6 6000
- #define SEC_7 7000
- #define SEC_8 8000
- #define SEC_9 9000
- #define SEC_10 10000
- #define SEC_15 15000
- #define SEC_20 20000
- #define SEC_30 30000

- #define MIN_1 60000
- #define NOTE_WHOLE 1000
- #define NOTE_HALF (NOTE_WHOLE/2)
- #define NOTE_QUARTER (NOTE_WHOLE/4)
- #define NOTE_EIGHT (NOTE_WHOLE/8)
- #define NOTE_SIXTEEN (NOTE_WHOLE/16)

5.11.1 Detailed Description

Constants for use with the Wait() function.

See Also

Wait()

5.11.2 Macro Definition Documentation

5.11.2.1 #define MIN_1 60000

1 minute

Examples:

ex_stat.spc.

5.11.2.2 #define MS_1 1

1 millisecond

Examples:

ex_arrays.spc.

5.11.2.3 #define MS_10 10

10 milliseconds

Examples:

ex_DAC0ToA3.spc.

5.11.2.4 #define MS_100 100

100 milliseconds

Examples:

ex_CurrentTick.spc.

5.11.2.5 #define MS_150 150

150 milliseconds

5.11.2.6 #define MS_2 2

2 milliseconds

5.11.2.7 #define MS_20 20

20 milliseconds

5.11.2.8 #define MS_200 200

200 milliseconds

5.11.2.9 #define MS_250 250

250 milliseconds

5.11.2.10 #define MS_3 3

3 milliseconds

5.11.2.11 #define MS_30 30

30 milliseconds

5.11.2.12 #define MS_300 300

300 milliseconds

5.11.2.13 #define MS_350 350

350 milliseconds

5.11.2.14 #define MS_4 4

4 milliseconds

5.11.2.15 #define MS_40 40

40 milliseconds

5.11.2.16 #define MS_400 400

400 milliseconds

5.11.2.17 #define MS_450 450

450 milliseconds

5.11.2.18 #define MS_5 5

5 milliseconds

5.11.2.19 #define MS_50 50

50 milliseconds

5.11.2.20 #define MS_500 500

500 milliseconds

Examples:

ex_serialout.spc, and t1.spc.

5.11.2.21 #define MS_6 6

6 milliseconds

5.11.2.22 #define MS_60 60

60 milliseconds

5.11.2.23 #define MS_600 600

600 milliseconds

5.11.2.24 #define MS_7 7

7 milliseconds

5.11.2.25 #define MS_70 70

70 milliseconds

5.11.2.26 #define MS_700 700

700 milliseconds

5.11.2.27 #define MS_8 8

8 milliseconds

5.11.2.28 #define MS_80 80

80 milliseconds

5.11.2.29 #define MS_800 800

800 milliseconds

5.11.2.30 #define MS_9 9

9 milliseconds

5.11.2.31 #define MS_90 90

90 milliseconds

5.11.2.32 #define MS_900 900

900 milliseconds

5.11.2.33 #define NOTE_EIGHT (NOTE_WHOLE/8)

The duration of an eighth note (ms)

5.11.2.34 #define NOTE_HALF (NOTE_WHOLE/2)

The duration of a half note (ms)

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

5.11.2.35 #define NOTE_QUARTER (NOTE_WHOLE/4)

The duration of a quarter note (ms)

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

5.11.2.36 #define NOTE_SIXTEEN (NOTE_WHOLE/16)

The duration of an sixteenth note (ms)

5.11.2.37 #define NOTE_WHOLE 1000

The duration of a whole note (ms)

5.11.2.38 #define SEC_1 1000

1 second

Examples:

ex_ledcontrol.spc, ex_Twinkle.spc, and ex_TwinkleL.spc.

5.11.2.39 #define SEC_10 10000

10 seconds

Examples:

ex_Stop.spc.

5.11.2.40 #define SEC_15 15000

15 seconds

Examples:

ex_timer.spc.

5.11.2.41 #define SEC_2 2000

2 seconds

Examples:

ex_timer.spc.

5.11.2.42 #define SEC_20 20000

20 seconds

5.11.2.43 #define SEC_3 3000

3 seconds

Examples:

ex_StartTask.spc.

5.11.2.44 #define SEC_30 30000

30 seconds

Examples:

ex_run.spc, ex_StopProcesses.spc, and ex_timer.spc.

5.11.2.45 #define SEC_4 4000

4 seconds

5.11.2.46 #define SEC_5 5000

5 seconds

Examples:

ex_ctype.spc, ex_StartTask.spc, ex_systemclock.spc, and ex_wait.spc.

5.11.2.47 #define SEC_6 6000

6 seconds

5.11.2.48 #define SEC_7 7000

7 seconds

5.11.2.49 #define SEC_8 8000

8 seconds

5.11.2.50 #define SEC_9 9000

9 seconds

5.12 Tone constants

Constants for use with the analog output frequency fields.

Macros

- #define TONE_C3 131
- #define TONE_CS3 139
- #define TONE_D3 147
- #define TONE_DS3 156
- #define TONE_E3 165
- #define TONE_F3 175
- #define TONE_FS3 185
- #define TONE_G3 196
- #define TONE_GS3 208
- #define TONE_A3 220
- #define TONE_AS3 233
- #define TONE_B3 247
- #define TONE_C4 262
- #define TONE_CS4 277
- #define TONE_D4 294
- #define TONE_DS4 311
- #define TONE_E4 330
- #define TONE_F4 349
- #define TONE_FS4 370
- #define TONE_G4 392
- #define TONE_GS4 415
- #define TONE_A4 440
- #define TONE_AS4 466
- #define TONE_B4 494
- #define TONE_C5 523
- #define TONE_CS5 554
- #define TONE_D5 587
- #define TONE_DS5 622
- #define TONE_E5 659
- #define TONE_F5 698
- #define TONE_FS5 740
- #define TONE_G5 784
- #define TONE_GS5 831
- #define TONE_A5 880
- #define TONE_AS5 932
- #define TONE_B5 988
- #define TONE_C6 1047
- #define TONE_CS6 1109
- #define TONE_D6 1175
- #define TONE_DS6 1245
- #define TONE_E6 1319
- #define TONE_F6 1397
- #define TONE_FS6 1480
- #define TONE_G6 1568

- #define TONE_GS6 1661
- #define TONE_A6 1760
- #define TONE_AS6 1865
- #define TONE_B6 1976
- #define TONE_C7 2093
- #define TONE_CS7 2217
- #define TONE_D7 2349
- #define TONE_DS7 2489
- #define TONE_E7 2637
- #define TONE_F7 2794
- #define TONE_FS7 2960
- #define TONE_G7 3136
- #define TONE_GS7 3322
- #define TONE_A7 3520
- #define TONE_AS7 3729
- #define TONE_B7 3951

5.12.1 Detailed Description

Constants for use with the analog output frequency fields.

See Also

DAC0Frequency, DAC1Frequency

5.12.2 Macro Definition Documentation

5.12.2.1 #define TONE_A3 220

Third octave A

Examples:

ex_Twinkle.spc.

5.12.2.2 #define TONE_A4 440

Fourth octave A

Examples:

ex_TwinkleL.spc.

5.12.2.3 #define TONE_A5 880

Fifth octave A

5.12.2.4 #define TONE_A6 1760

Sixth octave A

5.12.2.5 #define TONE_A7 3520

Seventh octave A

5.12.2.6 #define TONE_AS3 233

Third octave A sharp

5.12.2.7 #define TONE_AS4 466

Fourth octave A sharp

5.12.2.8 #define TONE_AS5 932

Fifth octave A sharp

5.12.2.9 #define TONE_AS6 1865

Sixth octave A sharp

5.12.2.10 #define TONE_AS7 3729

Seventh octave A sharp

5.12.2.11 #define TONE_B3 247

Third octave B

5.12.2.12 #define TONE_B4 494

Fourth octave B

5.12.2.13 #define TONE_B5 988

Fifth octave B

5.12.2.14 #define TONE_B6 1976

Sixth octave B

5.12.2.15 #define TONE_B7 3951

Seventh octave B

5.12.2.16 #define TONE_C3 131

Third octave C

Examples:

ex_Twinkle.spc.

5.12.2.17 #define TONE_C4 262

Fourth octave C

Examples:

ex_TwinkleL.spc.

5.12.2.18 #define TONE_C5 523

Fifth octave C

5.12.2.19 #define TONE_C6 1047

Sixth octave C

5.12.2.20 #define TONE_C7 2093

Seventh octave C

5.12.2.21 #define TONE_CS3 139

Third octave C sharp

5.12.2.22 #define TONE_CS4 277

Fourth octave C sharp

5.12.2.23 #define TONE_CS5 554

Fifth octave C sharp

5.12.2.24 #define TONE_CS6 1109

Sixth octave C sharp

5.12.2.25 #define TONE_CS7 2217

Seventh octave C sharp

5.12.2.26 #define TONE_D3 147

Third octave D

Examples:

ex_Twinkle.spc.

5.12.2.27 #define TONE_D4 294

Fourth octave D

Examples:

ex_TwinkleL.spc.

5.12.2.28 #define TONE_D5 587

Fifth octave D

5.12.2.29 #define TONE_D6 1175

Sixth octave D

5.12.2.30 #define TONE_D7 2349

Seventh octave D

5.12.2.31 #define TONE_DS3 156

Third octave D sharp

5.12.2.32 #define TONE_DS4 311

Fourth octave D sharp

5.12.2.33 #define TONE_DS5 622

Fifth octave D sharp

5.12.2.34 #define TONE_DS6 1245

Sixth octave D sharp

5.12.2.35 #define TONE_DS7 2489

Seventh octave D sharp

5.12.2.36 #define TONE_E3 165

Third octave E

Examples:

ex_Twinkle.spc.

5.12.2.37 #define TONE_E4 330

Fourth octave E

Examples:

ex_TwinkleL.spc.

5.12.2.38 #define TONE_E5 659

Fifth octave E

5.12.2.39 #define TONE_E6 1319

Sixth octave E

5.12.2.40 #define TONE_E7 2637

Seventh octave E

5.12.2.41 #define TONE_F3 175

Third octave F

Examples:

ex_Twinkle.spc.

5.12.2.42 #define TONE_F4 349

Fourth octave F

Examples:

ex_TwinkleL.spc.

5.12.2.43 #define TONE_F5 698

Fifth octave F

5.12.2.44 #define TONE_F6 1397

Sixth octave F

5.12.2.45 #define TONE_F7 2794

Seventh octave F

5.12.2.46 #define TONE_FS3 185

Third octave F sharp

5.12.2.47 #define TONE_FS4 370

Fourth octave F sharp

5.12.2.48 #define TONE_FS5 740

Fifth octave F sharp

5.12.2.49 #define TONE_FS6 1480

Sixth octave F sharp

5.12.2.50 #define TONE_FS7 2960

Seventh octave F sharp

5.12.2.51 #define TONE_G3 196

Third octave G

Examples:

ex_Twinkle.spc.

5.12.2.52 #define TONE_G4 392

Fourth octave G

Examples:

ex_TwinkleL.spc.

5.12.2.53 #define TONE_G5 784

Fifth octave G

5.12.2.54 #define TONE_G6 1568

Sixth octave G

5.12.2.55 #define TONE_G7 3136

Seventh octave G

5.12.2.56 #define TONE_GS3 208

Third octave G sharp

5.12.2.57 #define TONE_GS4 415

Fourth octave G sharp

5.12.2.58 #define TONE_GS5 831

Fifth octave G sharp

5.12.2.59 #define TONE_GS6 1661

Sixth octave G sharp

5.12.2.60 #define TONE_GS7 3322

Seventh octave G sharp

5.13 SuperPro C API

Functions which comprise the SuperPro C application programming interface.

Functions

- void Wait (long ms)

    Wait some milliseconds.
- void Yield (void)

    Yield to another task.
- void StopAllTasks (void)

    Stop all tasks.
- void Stop (bool bvalue)

    Stop the running program.
- void ExitTo (task newTask)

    Exit to another task.
- void StartTask (task t)

    Start a task.
- int SizeOf (variant &value)

    Calculate the size of a variable.
- int read (void)

    Read a value from a file.
- int write (const int value)

    Write value to file.
- int sqrt (int x)

    Compute square root.
- int abs (int num)

    Absolute value.
- char sign (int num)

    Sign value.
- int close (void)

    Close file.
- byte open (const char *mode)

    Open file.
- char putchar (const char ch)

    Write character to debug device.
- int puts (const char *str)

    Write string to debug device.
- void printf (const char *format,...)

    Print formatted data to debug device.
- void abort (void)

    Abort current process.
- long CurrentTick (void)

    Read the current system tick.
- int pop (void)

    Pop a value off the stack.
- int push (int value)

Push a value onto the stack.

- void RotateLeft (int &value)

  Rotate left.
- void RotateRight (int &value)

  Rotate right.
- void Run (const int slot)

  Run another program.
- int stat (void)

  Check log file status.
- void StopProcesses (void)

  Stop all processes.

5.13.1 Detailed Description

Functions which comprise the SuperPro C application programming interface.

5.13.2 Function Documentation

5.13.2.1 void abort (void) [inline]

Abort current process.

Aborts the process with an abnormal program termination. The function never returns to its caller.

Examples:

ex_abort.spc.

5.13.2.2 int abs (int *num*) [inline]

Absolute value.

Return the absolute value of the value argument. Any scalar type can be passed into this function.

Parameters

num	The numeric value.

Returns

The absolute value of num. The return type matches the input type.

Examples:

ex_abs.spc.

5.13.2.3 int close (void) [inline]

Close file.

Close the log file.

Returns

The result code.

Examples:

ex_close.spc, ex_open.spc, ex_read.spc, ex_stat.spc, and ex_write.spc.

5.13.2.4 long CurrentTick (void) [inline]

Read the current system tick.

This function lets you current system tick count.

Returns

The current system tick count.

Examples:

ex_CurrentTick.spc, ex_run.spc, ex_sign.spc, ex_sqrt.spc, ex_StopAllTasks.spc, and ex_StopProcesses.spc.

5.13.2.5 void ExitTo (task *newTask*) [inline]

Exit to another task.

Immediately exit the current task and start executing the specified task.

Parameters

newTask	The task to start executing after exiting the current task.

Examples:

ex_exitto.spc.

5.13.2.6 byte open (const char * *mode*) [inline]

Open file.

Opens the log file. The operations that are allowed on the stream and how these are performed are defined by the mode parameter.

Parameters

mode	The file access mode. Valid values are "r" - opens the existing log file for reading, "w" - creates a new log file and opens it for writing.

Returns

The result code.

Examples:

ex_close.spc, ex_open.spc, ex_read.spc, ex_stat.spc, and ex_write.spc.

5.13.2.7 int pop (void) [inline]

Pop a value off the stack.

Pop a 32-bit integer value off the top of the stack.

Returns

The value popped off the top of the stack.

Examples:

ex_pushpop.spc.

5.13.2.8 void printf (const char * *format*, ...) [inline]

Print formatted data to debug device.

Writes to the debug device a sequence of data formatted as the format argument specifies. After the format parameter, the function expects a variable number of parameters.

Parameters

format	A constant string literal specifying the desired format.

Examples:

ex_abs.spc, ex_arrays.spc, ex_close.spc, ex_ctype.spc, ex_CurrentTick.spc, ex_DAC0ToA3.spc, ex_digiserial.spc, ex_exitto.spc, ex_isalnum.spc, ex_isalpha.spc, ex_iscntrl.spc, ex_isdigit.spc, ex_isgraph.spc, ex_islower.spc, ex_isprint.spc, ex_ispunct.spc, ex_isspace.spc, ex_isupper.spc, ex_isxdigit.spc, ex_open.spc, ex_printf.spc, ex_pushpop.spc, ex_read.spc, ex_rotate.spc, ex_serialout.spc, ex_sign.spc, ex_SizeOf.spc, ex_sqrt.spc, ex_systemclock.spc, ex_timer.spc, ex_tolower.spc, ex_toupper.spc, ex_wait.spc, and ex_yield.spc.

5.13.2.9 int push (int *value*) [inline]

Push a value onto the stack.

Push a 32-bit integer value onto the top of the stack.

Parameters

value	The value you want to push onto the stack.

Returns

The value pushed onto the stack.

Examples:

ex_pushpop.spc.

5.13.2.10 char putchar (const char *ch*) [inline]

Write character to debug device.

Writes a character to the debug device. If there are no errors, the same character that has been written is returned.

Parameters

ch	The character to be written.

Returns

The character written to the file.

Examples:

ex_putchar.spc.

5.13.2.11 int puts (const char * *str*) [inline]

Write string to debug device.

Writes the string to the debug device. The null terminating character at the end of the string is not written. If there are no errors, a non-negative value is returned.

Parameters

str	The string of characters to be written.

Returns

The result code.

Examples:

ex_digiserial.spc, ex_puts.spc, ex_StartTask.spc, and t1.spc.

5.13.2.12 int read (void) [inline]

Read a value from a file.

Read a value from the file associated with the specified handle. The handle parameter must be a variable. The value parameter must be a variable. The type of the value parameter determines the number of bytes of data read.

Returns

The function call result.

Examples:

ex_close.spc, ex_open.spc, and ex_read.spc.

5.13.2.13 void RotateLeft (int & *value*) [inline]

Rotate left.

Rotate the specified variable one bit left through carry.

Parameters

value	The value to rotate left one bit.

Examples:

ex_rotate.spc.

5.13.2.14 void RotateRight (int & *value*) [inline]

Rotate right.

Rotate the specified variable one bit right through carry.

Parameters

value	The value to rotate right one bit.

Examples:

ex_rotate.spc.

5.13.2.15 void Run (const int *slot*) [inline]

Run another program.

Run the program in the specified slot. The current program will terminate.

Parameters

slot	The constant slot number for the program you want to execute. See Program slot constants.

Examples:

ex_run.spc.

5.13.2.16 char sign (int *num*) [inline]

Sign value.

Return the sign of the value argument (-1, 0, or 1). Any scalar type can be passed into this function.

Parameters

num	The numeric value for which to calculate its sign value.

Returns

-1 if the parameter is negative, 0 if the parameter is zero, or 1 if the parameter is positive.

Examples:

ex_sign.spc.

5.13.2.17 int SizeOf (variant & *value*) [inline]

Calculate the size of a variable.

Calculate the number of bytes required to store the contents of the variable passed into the function.

Parameters

value	The variable.

Returns

The number of bytes occupied by the variable.

Examples:

ex_SizeOf.spc.

5.13.2.18 int sqrt (int *x*) [inline]

Compute square root.

Computes the square root of x.

Parameters

x	integer value.

Returns

Square root of x.

Examples:

ex_sqrt.spc.

5.13.2.19 void StartTask (task *t*) [inline]

Start a task.

Start the specified task.

Parameters

t	The task to start.

Examples:

ex_StartTask.spc.

5.13.2.20 int stat (void) [inline]

Check log file status.

Check the status of the system log file.

Returns

The log file status. See Log status constants.

Examples:

ex_stat.spc.

5.13.2.21 void Stop (bool *bvalue*) [inline]

Stop the running program.

Stop the running program if bvalue is true. This will halt the program completely, so any code following this command will be ignored.

Parameters

bvalue	If this value is true the program will stop executing.

Examples:

ex_Stop.spc.

5.13.2.22 void StopAllTasks (void) [inline]

Stop all tasks.

Stop all currently running tasks. This will halt the program completely, so any code following this command will be ignored.

Examples:

ex_StopAllTasks.spc.

5.13.2.23 void StopProcesses (void) [inline]

Stop all processes.

Stop all running tasks except for the main task.

Examples:

ex_StopProcesses.spc.

5.13.2.24 void Wait (long *ms*) [inline]

Wait some milliseconds.

Make a task sleep for specified amount of time (in 1000ths of a second).

Parameters

ms	The number of milliseconds to sleep.

Examples:

ex_arrays.spc, ex_ctype.spc, ex_CurrentTick.spc, ex_DAC0ToA3.spc, ex_digiserial.spc, ex_ledcontrol.spc, ex_-serialout.spc, ex_StartTask.spc, ex_systemclock.spc, ex_timer.spc, ex_Twinkle.spc, ex_TwinkleL.spc, ex_wait.spc, if_test.spc, and t1.spc.

5.13.2.25 int write (const int *value*) [inline]

Write value to file.

Write a value to the file associated with the specified handle. The handle parameter must be a variable. The value parameter must be a constant, a constant expression, or a variable. The type of the value parameter determines the number of bytes of data written.

Parameters

value	The value to write to the file.

Returns

The function call result.

Examples:

ex_stat.spc, and ex_write.spc.

5.13.2.26 void Yield (void) [inline]

Yield to another task.

Make a task yield to another concurrently running task.

Examples:

ex_yield.spc.

5.14 ctype API

Standard C ctype API functions.

Functions

- int isupper (int c)

  Check if character is uppercase letter.
- int islower (int c)

  Check if character is lowercase letter.
- int isalpha (int c)

  Check if character is alphabetic.
- int isdigit (int c)

  Check if character is decimal digit.
- int isalnum (int c)

  Check if character is alphanumeric.
- int isspace (int c)

  Check if character is a white-space.
- int iscntrl (int c)

  Check if character is a control character.
- int isprint (int c)

  Check if character is printable.
- int isgraph (int c)

  Check if character has graphical representation.
- int ispunct (int c)

  Check if character is a punctuation.
- int isxdigit (int c)

  Check if character is hexadecimal digit.
- int toupper (int c)

  Convert lowercase letter to uppercase.
- int tolower (int c)

  Convert uppercase letter to lowercase.

5.14.1 Detailed Description

Standard C ctype API functions.

5.14.2 Function Documentation

5.14.2.1 int isalnum (int *c*) `[inline]`

Check if character is alphanumeric.

Checks if parameter c is either a decimal digit or an uppercase or lowercase letter. The result is true if either isalpha or isdigit would also return true.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is either a digit or a letter, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isalnum.spc.

5.14.2.2 int isalpha (int *c*) [inline]

Check if character is alphabetic.

Checks if parameter c is either an uppercase or lowercase letter.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is an alphabetic letter, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isalpha.spc.

5.14.2.3 int iscntrl (int *c*) [inline]

Check if character is a control character.

Checks if parameter c is a control character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a control character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_iscntrl.spc.

5.14.2.4 int isdigit (int *c*) [inline]

Check if character is decimal digit.

Checks if parameter c is a decimal digit character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a decimal digit, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isdigit.spc.

5.14.2.5 int isgraph (int *c*) [inline]

Check if character has graphical representation.

Checks if parameter c is a character with a graphical representation.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c has a graphical representation, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isgraph.spc.

5.14.2.6 int islower (int *c*) [inline]

Check if character is lowercase letter.

Checks if parameter c is an lowercase alphabetic letter.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is an lowercase alphabetic letter, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_islower.spc.

5.14.2.7 int isprint (int *c*) [inline]

Check if character is printable.

Checks if parameter c is a printable character (i.e., not a control character).

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a printable character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isprint.spc.

5.14.2.8 int ispunct (int *c*) [inline]

Check if character is a punctuation.

Checks if parameter c is a punctuation character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a punctuation character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_ispunct.spc.

5.14.2.9 int isspace (int *c*) [inline]

Check if character is a white-space.

Checks if parameter c is a white-space character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a white-space character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isspace.spc.

5.14.2.10 int isupper (int *c*) [inline]

Check if character is uppercase letter.

Checks if parameter c is an uppercase alphabetic letter.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is an uppercase alphabetic letter, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isupper.spc.

5.14.2.11 int isxdigit (int *c*) [inline]

Check if character is hexadecimal digit.

Checks if parameter c is a hexadecimal digit character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a hexadecimal digit character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isxdigit.spc.

5.14.2.12 int tolower (int *c*) [inline]

Convert uppercase letter to lowercase.

Converts parameter c to its lowercase equivalent if c is an uppercase letter and has a lowercase equivalent. If no such conversion is possible, the value returned is c unchanged.

Parameters

c	Uppercase letter character to be converted.

Returns

The lowercase equivalent to c, if such value exists, or c (unchanged) otherwise..

Examples:

ex_ctype.spc, and ex_tolower.spc.

5.14.2.13 int toupper (int *c*) [inline]

Convert lowercase letter to uppercase.

Converts parameter c to its uppercase equivalent if c is a lowercase letter and has an uppercase equivalent. If no such conversion is possible, the value returned is c unchanged.

Parameters

c	Lowercase letter character to be converted.

Returns

The uppercase equivalent to c, if such value exists, or c (unchanged) otherwise..

Examples:

ex_ctype.spc, and ex_toupper.spc.

6 File Documentation

6.1 SPCAPIDocs.h File Reference

Additional documentation for the SPC API.

```
#include "SPCDefs.h"
```

6.1.1 Detailed Description

Additional documentation for the SPC API. SPCAPIDocs.h contains additional documentation for the SPC API

License:

Author

John Hansen (bricxcc_at_comcast.net)

Date

2013-02-20

Version

4

6.2 SPCDefs.h File Reference

Constants, macros, and API functions for SPC.

```
#include "spmem.h"
```

Macros

- #define TRUE 1
- #define FALSE 0
- #define SERIAL_BUFFER_SIZE 255
- #define CHAR_BIT 32
- #define LONG_MIN -2147483648
- #define SCHAR_MIN -2147483648
- #define INT_MIN -2147483648
- #define CHAR_MIN -2147483648
- #define LONG_MAX 2147483647
- #define SCHAR_MAX 2147483647
- #define INT_MAX 2147483647

- #define CHAR_MAX 2147483647
- #define DAC_MODE_DCOUT 0
- #define DAC_MODE_SINEWAVE 1
- #define DAC_MODE_SQUAREWAVE 2
- #define DAC_MODE_SAWPOSWAVE 3
- #define DAC_MODE_SAWNEGWAVE 4
- #define DAC_MODE_TRIANGLEWAVE 5
- #define DAC_MODE_PWMVOLTAGE 6
- #define DAC_MODE_RESTART_MASK 0x80
- #define LED_BLUE 0x02
- #define LED_RED 0x01
- #define DIGI_PIN0 0x01
- #define DIGI_PIN1 0x02
- #define DIGI_PIN2 0x04
- #define DIGI_PIN3 0x08
- #define DIGI_PIN4 0x10
- #define DIGI_PIN5 0x20
- #define DIGI_PIN6 0x40
- #define DIGI_PIN7 0x80
- #define STROBE_S0 0x01
- #define STROBE_S1 0x02
- #define STROBE_S2 0x04
- #define STROBE_S3 0x08
- #define STROBE_READ 0x10
- #define STROBE_WRITE 0x20
- #define SLOT1 0
- #define SLOT2 1
- #define SLOT3 2
- #define SLOT4 3
- #define SLOT5 4
- #define SLOT6 5
- #define SLOT7 6
- #define LOG_STATUS_OPEN 2
- #define LOG_STATUS_BUSY 1
- #define LOG_STATUS_CLOSED 0
- #define MS_1 1
- #define MS_2 2
- #define MS_3 3
- #define MS_4 4
- #define MS_5 5
- #define MS_6 6
- #define MS_7 7
- #define MS_8 8
- #define MS_9 9
- #define MS_10 10
- #define MS_20 20
- #define MS_30 30
- #define MS_40 40
- #define MS_50 50
- #define MS_60 60
- #define MS_70 70

- #define MS_80 80
- #define MS_90 90
- #define MS_100 100
- #define MS_150 150
- #define MS_200 200
- #define MS_250 250
- #define MS_300 300
- #define MS_350 350
- #define MS_400 400
- #define MS_450 450
- #define MS_500 500
- #define MS_600 600
- #define MS_700 700
- #define MS_800 800
- #define MS_900 900
- #define SEC_1 1000
- #define SEC_2 2000
- #define SEC_3 3000
- #define SEC_4 4000
- #define SEC_5 5000
- #define SEC_6 6000
- #define SEC_7 7000
- #define SEC_8 8000
- #define SEC_9 9000
- #define SEC_10 10000
- #define SEC_15 15000
- #define SEC_20 20000
- #define SEC_30 30000
- #define MIN_1 60000
- #define NOTE_WHOLE 1000
- #define NOTE_HALF (NOTE_WHOLE/2)
- #define NOTE_QUARTER (NOTE_WHOLE/4)
- #define NOTE_EIGHT (NOTE_WHOLE/8)
- #define NOTE_SIXTEEN (NOTE_WHOLE/16)
- #define TONE_C3 131
- #define TONE_CS3 139
- #define TONE_D3 147
- #define TONE_DS3 156
- #define TONE_E3 165
- #define TONE_F3 175
- #define TONE_FS3 185
- #define TONE_G3 196
- #define TONE_GS3 208
- #define TONE_A3 220
- #define TONE_AS3 233
- #define TONE_B3 247
- #define TONE_C4 262
- #define TONE_CS4 277
- #define TONE_D4 294
- #define TONE_DS4 311
- #define TONE_E4 330

- #define TONE_F4 349
- #define TONE_FS4 370
- #define TONE_G4 392
- #define TONE_GS4 415
- #define TONE_A4 440
- #define TONE_AS4 466
- #define TONE_B4 494
- #define TONE_C5 523
- #define TONE_CS5 554
- #define TONE_D5 587
- #define TONE_DS5 622
- #define TONE_E5 659
- #define TONE_F5 698
- #define TONE_FS5 740
- #define TONE_G5 784
- #define TONE_GS5 831
- #define TONE_A5 880
- #define TONE_AS5 932
- #define TONE_B5 988
- #define TONE_C6 1047
- #define TONE_CS6 1109
- #define TONE_D6 1175
- #define TONE_DS6 1245
- #define TONE_E6 1319
- #define TONE_F6 1397
- #define TONE_FS6 1480
- #define TONE_G6 1568
- #define TONE_GS6 1661
- #define TONE_A6 1760
- #define TONE_AS6 1865
- #define TONE_B6 1976
- #define TONE_C7 2093
- #define TONE_CS7 2217
- #define TONE_D7 2349
- #define TONE_DS7 2489
- #define TONE_E7 2637
- #define TONE_F7 2794
- #define TONE_FS7 2960
- #define TONE_G7 3136
- #define TONE_GS7 3322
- #define TONE_A7 3520
- #define TONE_AS7 3729
- #define TONE_B7 3951

Functions

- void Wait (long ms)

    Wait some milliseconds.
- void Yield (void)

    Yield to another task.
- void StopAllTasks (void)

    Stop all tasks.
- void Stop (bool bvalue)

    Stop the running program.
- void ExitTo (task newTask)

    Exit to another task.
- void StartTask (task t)

    Start a task.
- int SizeOf (variant &value)

    Calculate the size of a variable.
- int read (void)

    Read a value from a file.
- int write (const int value)

    Write value to file.
- int sqrt (int x)

    Compute square root.
- int abs (int num)

    Absolute value.
- char sign (int num)

    Sign value.
- int close (void)

    Close file.
- byte open (const char ∗mode)

    Open file.
- char putchar (const char ch)

    Write character to debug device.
- int puts (const char ∗str)

    Write string to debug device.
- void printf (const char ∗format,...)

    Print formatted data to debug device.
- void abort (void)

    Abort current process.
- long CurrentTick (void)

    Read the current system tick.
- int pop (void)

    Pop a value off the stack.
- int push (int value)

    Push a value onto the stack.
- void RotateLeft (int &value)

    Rotate left.
- void RotateRight (int &value)

Rotate right.

- void Run (const int slot)

  Run another program.
- int stat (void)

  Check log file status.
- void StopProcesses (void)

  Stop all processes.
- int isupper (int c)

  Check if character is uppercase letter.
- int islower (int c)

  Check if character is lowercase letter.
- int isalpha (int c)

  Check if character is alphabetic.
- int isdigit (int c)

  Check if character is decimal digit.
- int isalnum (int c)

  Check if character is alphanumeric.
- int isspace (int c)

  Check if character is a white-space.
- int iscntrl (int c)

  Check if character is a control character.
- int isprint (int c)

  Check if character is printable.
- int isgraph (int c)

  Check if character has graphical representation.
- int ispunct (int c)

  Check if character is a punctuation.
- int isxdigit (int c)

  Check if character is hexadecimal digit.
- int toupper (int c)

  Convert lowercase letter to uppercase.
- int tolower (int c)

  Convert uppercase letter to lowercase.

6.2.1 Detailed Description

Constants, macros, and API functions for SPC. SPCDefs.h contains declarations for the SPC API resources

License:

Author

John Hansen (bricxcc_at_comcast.net)

Date

2013-02-20

Version

3

6.2.2 Macro Definition Documentation

6.2.2.1 #define CHAR_BIT 32

The number of bits in the char type

6.2.2.2 #define CHAR_MAX 2147483647

The maximum value of the char type

6.2.2.3 #define CHAR_MIN -2147483648

The minimum value of the char type

6.2.2.4 #define DAC_MODE_DCOUT 0

Steady (DC) voltage output.

6.2.2.5 #define DAC_MODE_PWMVOLTAGE 6

PWM square wave output.

6.2.2.6 #define DAC_MODE_RESTART_MASK 0x80

Add mask to DAC mode constants to force waveform generation from the start of the wave table.

6.2.2.7 #define DAC_MODE_SAWNEGWAVE 4

Negative going sawtooth output.

6.2.2.8 #define DAC_MODE_SAWPOSWAVE 3

Positive going sawtooth output.

6.2.2.9 #define DAC_MODE_SINEWAVE 1

Sine wave output.

Examples:

ex_DAC0ToA3.spc.

6.2.2.10 #define DAC_MODE_SQUAREWAVE 2

Square wave output.

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

6.2.2.11 #define DAC_MODE_TRIANGLEWAVE 5

Triangle wave output.

6.2.2.12 #define DIGI_PIN0 0x01

Access digital pin 0 (B0)

6.2.2.13 #define DIGI_PIN1 0x02

Access digital pin 1 (B1)

6.2.2.14 #define DIGI_PIN2 0x04

Access digital pin 2 (B2)

6.2.2.15 #define DIGI_PIN3 0x08

Access digital pin 3 (B3)

6.2.2.16 #define DIGI_PIN4 0x10

Access digital pin 4 (B4)

6.2.2.17 #define DIGI_PIN5 0x20

Access digital pin 5 (B5)

6.2.2.18 #define DIGI_PIN6 0x40

Access digital pin 6 (B6)

6.2.2.19 #define DIGI_PIN7 0x80

Access digital pin 7 (B7)

6.2.2.20 #define FALSE 0

A false value

6.2.2.21 #define INT_MAX 2147483647

The maximum value of the int type

6.2.2.22 #define INT_MIN -2147483648

The minimum value of the int type

6.2.2.23 #define LED_BLUE 0x02

Turn on the blue onboard LED.

Examples:

ex_ledcontrol.spc.

6.2.2.24 #define LED_RED 0x01

Turn on the red onboard LED.

Examples:

ex_ledcontrol.spc.

6.2.2.25 #define LOG_STATUS_BUSY 1

Log file is busy.

6.2.2.26 #define LOG_STATUS_CLOSED 0

Log file is closed.

6.2.2.27 #define LOG_STATUS_OPEN 2

Log file is open.

Examples:

ex_stat.spc.

6.2.2.28 #define LONG_MAX 2147483647

The maximum value of the long type

6.2.2.29 #define LONG_MIN -2147483648

The minimum value of the long type

6.2.2.30 #define MIN_1 60000

1 minute

Examples:

ex_stat.spc.

6.2.2.31 #define MS_1 1

1 millisecond

Examples:

ex_arrays.spc.

6.2.2.32 #define MS_10 10

10 milliseconds

Examples:

ex_DAC0ToA3.spc.

6.2.2.33 #define MS_100 100

100 milliseconds

Examples:

ex_CurrentTick.spc.

6.2.2.34 #define MS_150 150

150 milliseconds

6.2.2.35 #define MS_2 2

2 milliseconds

6.2.2.36 #define MS_20 20

20 milliseconds

6.2.2.37 #define MS_200 200

200 milliseconds

6.2.2.38 #define MS_250 250

250 milliseconds

6.2.2.39 #define MS_3 3

3 milliseconds

6.2.2.40 #define MS_30 30

30 milliseconds

6.2.2.41 #define MS_300 300

300 milliseconds

6.2.2.42 #define MS_350 350

350 milliseconds

6.2.2.43 #define MS_4 4

4 milliseconds

6.2.2.44 #define MS_40 40

40 milliseconds

6.2.2.45 #define MS_400 400

400 milliseconds

6.2.2.46 #define MS_450 450

450 milliseconds

6.2.2.47 #define MS_5 5

5 milliseconds

6.2.2.48 #define MS_50 50

50 milliseconds

6.2.2.49 #define MS_500 500

500 milliseconds

Examples:

ex_serialout.spc, and t1.spc.

6.2.2.50 #define MS_6 6

6 milliseconds

6.2.2.51 #define MS_60 60

60 milliseconds

6.2.2.52 #define MS_600 600

600 milliseconds

6.2.2.53 #define MS_7 7

7 milliseconds

6.2.2.54 #define MS_70 70

70 milliseconds

6.2.2.55 #define MS_700 700

700 milliseconds

6.2.2.56 #define MS_8 8

8 milliseconds

6.2.2.57 #define MS_80 80

80 milliseconds

6.2.2.58 #define MS_800 800

800 milliseconds

6.2.2.59 #define MS_9 9

9 milliseconds

6.2.2.60 #define MS_90 90

90 milliseconds

6.2.2.61 #define MS_900 900

900 milliseconds

6.2.2.62 #define NOTE_EIGHT (NOTE_WHOLE/8)

The duration of an eighth note (ms)

6.2.2.63 #define NOTE_HALF (NOTE_WHOLE/2)

The duration of a half note (ms)

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

6.2.2.64 #define NOTE_QUARTER (NOTE_WHOLE/4)

The duration of a quarter note (ms)

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

6.2.2.65 #define NOTE_SIXTEEN (NOTE_WHOLE/16)

The duration of an sixteenth note (ms)

6.2.2.66 #define NOTE_WHOLE 1000

The duration of a whole note (ms)

6.2.2.67 #define SCHAR_MAX 2147483647

The maximum value of the signed char type

6.2.2.68 #define SCHAR_MIN -2147483648

The minimum value of the signed char type

6.2.2.69 #define SEC_1 1000

1 second

Examples:

ex_ledcontrol.spc, ex_Twinkle.spc, and ex_TwinkleL.spc.

6.2.2.70 #define SEC_10 10000

10 seconds

Examples:

ex_Stop.spc.

6.2.2.71 #define SEC_15 15000

15 seconds

Examples:

ex_timer.spc.

6.2.2.72 #define SEC_2 2000

2 seconds

Examples:

ex_timer.spc.

6.2.2.73 #define SEC_20 20000

20 seconds

6.2.2.74 #define SEC_3 3000

3 seconds

Examples:

ex_StartTask.spc.

6.2.2.75 #define SEC_30 30000

30 seconds

Examples:

ex_run.spc, ex_StopProcesses.spc, and ex_timer.spc.

6.2.2.76 #define SEC_4 4000

4 seconds

6.2.2.77 #define SEC_5 5000

5 seconds

Examples:

ex_ctype.spc, ex_StartTask.spc, ex_systemclock.spc, and ex_wait.spc.

6.2.2.78 #define SEC_6 6000

6 seconds

6.2.2.79 #define SEC_7 7000

7 seconds

6.2.2.80 #define SEC_8 8000

8 seconds

6.2.2.81 #define SEC_9 9000

9 seconds

6.2.2.82 #define SERIAL_BUFFER_SIZE 255

Serial port receive and send buffer size

6.2.2.83 #define SLOT1 0

Program slot 1.

6.2.2.84 #define SLOT2 1

Program slot 2.

Examples:

ex_run.spc.

6.2.2.85 #define SLOT3 2

Program slot 3.

6.2.2.86 #define SLOT4 3

Program slot 4.

6.2.2.87 #define SLOT5 4

Program slot 5.

6.2.2.88 #define SLOT6 5

Program slot 6.

6.2.2.89 #define SLOT7 6

Program slot 7.

6.2.2.90 #define STROBE_READ 0x10

Access read pin (RD)

6.2.2.91 #define STROBE_S0 0x01

Access strobe 0 pin (S0)

6.2.2.92 #define STROBE_S1 0x02

Access strobe 1 pin (S1)

6.2.2.93 #define STROBE_S2 0x04

Access strobe 2 pin (S2)

6.2.2.94 #define STROBE_S3 0x08

Access strobe 3 pin (S3)

6.2.2.95 #define STROBE_WRITE 0x20

Access write pin (WR)

6.2.2.96 #define TONE_A3 220

Third octave A

Examples:

ex_Twinkle.spc.

6.2.2.97 #define TONE_A4 440

Fourth octave A

Examples:

ex_TwinkleL.spc.

6.2.2.98 #define TONE_A5 880

Fifth octave A

6.2.2.99 #define TONE_A6 1760

Sixth octave A

6.2.2.100 #define TONE_A7 3520

Seventh octave A

6.2.2.101 #define TONE_AS3 233

Third octave A sharp

6.2.2.102 #define TONE_AS4 466

Fourth octave A sharp

6.2.2.103 #define TONE_AS5 932

Fifth octave A sharp

6.2.2.104 #define TONE_AS6 1865

Sixth octave A sharp

6.2.2.105 #define TONE_AS7 3729

Seventh octave A sharp

6.2.2.106 #define TONE_B3 247

Third octave B

6.2.2.107 #define TONE_B4 494

Fourth octave B

6.2.2.108 #define TONE_B5 988

Fifth octave B

6.2.2.109 #define TONE_B6 1976

Sixth octave B

6.2.2.110 #define TONE_B7 3951

Seventh octave B

6.2.2.111 #define TONE_C3 131

Third octave C

Examples:

ex_Twinkle.spc.

6.2.2.112 #define TONE_C4 262

Fourth octave C

Examples:

ex_TwinkleL.spc.

6.2.2.113 #define TONE_C5 523

Fifth octave C

6.2.2.114 #define TONE_C6 1047

Sixth octave C

6.2.2.115 #define TONE_C7 2093

Seventh octave C

6.2.2.116 #define TONE_CS3 139

Third octave C sharp

6.2.2.117 #define TONE_CS4 277

Fourth octave C sharp

6.2.2.118 #define TONE_CS5 554

Fifth octave C sharp

6.2.2.119 #define TONE_CS6 1109

Sixth octave C sharp

6.2.2.120 #define TONE_CS7 2217

Seventh octave C sharp

6.2.2.121 #define TONE_D3 147

Third octave D

Examples:

ex_Twinkle.spc.

6.2.2.122 #define TONE_D4 294

Fourth octave D

Examples:

ex_TwinkleL.spc.

6.2.2.123 #define TONE_D5 587

Fifth octave D

6.2.2.124 #define TONE_D6 1175

Sixth octave D

6.2.2.125 #define TONE_D7 2349

Seventh octave D

6.2.2.126 #define TONE_DS3 156

Third octave D sharp

6.2.2.127 #define TONE_DS4 311

Fourth octave D sharp

6.2.2.128 #define TONE_DS5 622

Fifth octave D sharp

6.2.2.129 #define TONE_DS6 1245

Sixth octave D sharp

6.2.2.130 #define TONE_DS7 2489

Seventh octave D sharp

6.2.2.131 #define TONE_E3 165

Third octave E

Examples:

ex_Twinkle.spc.

6.2.2.132 #define TONE_E4 330

Fourth octave E

Examples:

ex_TwinkleL.spc.

6.2.2.133 #define TONE_E5 659

Fifth octave E

6.2.2.134 #define TONE_E6 1319

Sixth octave E

6.2.2.135 #define TONE_E7 2637

Seventh octave E

6.2.2.136 #define TONE_F3 175

Third octave F

Examples:

ex_Twinkle.spc.

6.2.2.137 #define TONE_F4 349

Fourth octave F

Examples:

ex_TwinkleL.spc.

6.2.2.138 #define TONE_F5 698

Fifth octave F

6.2.2.139 #define TONE_F6 1397

Sixth octave F

6.2.2.140 #define TONE_F7 2794

Seventh octave F

6.2.2.141 #define TONE_FS3 185

Third octave F sharp

6.2.2.142 #define TONE_FS4 370

Fourth octave F sharp

6.2.2.143 #define TONE_FS5 740

Fifth octave F sharp

6.2.2.144 #define TONE_FS6 1480

Sixth octave F sharp

6.2.2.145 #define TONE_FS7 2960

Seventh octave F sharp

6.2.2.146 #define TONE_G3 196

Third octave G

Examples:

ex_Twinkle.spc.

6.2.2.147 #define TONE_G4 392

Fourth octave G

Examples:

ex_TwinkleL.spc.

6.2.2.148 #define TONE_G5 784

Fifth octave G

6.2.2.149 #define TONE_G6 1568

Sixth octave G

6.2.2.150 #define TONE_G7 3136

Seventh octave G

6.2.2.151 #define TONE_GS3 208

Third octave G sharp

6.2.2.152 #define TONE_GS4 415

Fourth octave G sharp

6.2.2.153 #define TONE_GS5 831

Fifth octave G sharp

6.2.2.154 #define TONE_GS6 1661

Sixth octave G sharp

6.2.2.155 #define TONE_GS7 3322

Seventh octave G sharp

6.2.2.156 #define TRUE 1

A true value

6.2.3 Function Documentation

6.2.3.1 void abort (void) [inline]

Abort current process.

Aborts the process with an abnormal program termination. The function never returns to its caller.

Examples:

ex_abort.spc.

6.2.3.2 int abs (int *num*) [inline]

Absolute value.

Return the absolute value of the value argument. Any scalar type can be passed into this function.

Parameters

num	The numeric value.

Returns

The absolute value of num. The return type matches the input type.

Examples:

ex_abs.spc.

6.2.3.3 int close (void) [inline]

Close file.

Close the log file.

Returns

The result code.

Examples:

ex_close.spc, ex_open.spc, ex_read.spc, ex_stat.spc, and ex_write.spc.

6.2.3.4 long CurrentTick (void) [inline]

Read the current system tick.

This function lets you current system tick count.

Returns

The current system tick count.

Examples:

ex_CurrentTick.spc, ex_run.spc, ex_sign.spc, ex_sqrt.spc, ex_StopAllTasks.spc, and ex_StopProcesses.spc.

6.2.3.5 void ExitTo (task *newTask*) [inline]

Exit to another task.

Immediately exit the current task and start executing the specified task.

Parameters

newTask	The task to start executing after exiting the current task.

Examples:

ex_exitto.spc.

6.2.3.6 int isalnum (int *c*) [inline]

Check if character is alphanumeric.

Checks if parameter c is either a decimal digit or an uppercase or lowercase letter. The result is true if either isalpha or isdigit would also return true.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is either a digit or a letter, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isalnum.spc.

6.2.3.7 int isalpha (int *c*) [inline]

Check if character is alphabetic.

Checks if parameter c is either an uppercase or lowercase letter.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is an alphabetic letter, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isalpha.spc.

6.2.3.8 int iscntrl (int *c*) [inline]

Check if character is a control character.

Checks if parameter c is a control character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a control character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_iscntrl.spc.

6.2.3.9 int isdigit (int *c*) [inline]

Check if character is decimal digit.

Checks if parameter c is a decimal digit character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a decimal digit, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isdigit.spc.

6.2.3.10 int isgraph (int *c*) [inline]

Check if character has graphical representation.

Checks if parameter c is a character with a graphical representation.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c has a graphical representation, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isgraph.spc.

6.2.3.11 int islower (int *c*) [inline]

Check if character is lowercase letter.

Checks if parameter c is an lowercase alphabetic letter.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is an lowercase alphabetic letter, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_islower.spc.

6.2.3.12 int isprint (int *c*) [inline]

Check if character is printable.

Checks if parameter c is a printable character (i.e., not a control character).

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a printable character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isprint.spc.

6.2.3.13 int ispunct (int *c*) [inline]

Check if character is a punctuation.

Checks if parameter c is a punctuation character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a punctuation character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_ispunct.spc.

6.2.3.14 int isspace (int *c*) [inline]

Check if character is a white-space.

Checks if parameter c is a white-space character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a white-space character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isspace.spc.

6.2.3.15 int isupper (int *c*) [inline]

Check if character is uppercase letter.

Checks if parameter c is an uppercase alphabetic letter.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is an uppercase alphabetic letter, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isupper.spc.

6.2.3.16 int isxdigit (int *c*) [inline]

Check if character is hexadecimal digit.

Checks if parameter c is a hexadecimal digit character.

Parameters

c	Character to be checked.

Returns

Returns a non-zero value (true) if c is a hexadecimal digit character, otherwise it returns 0 (false).

Examples:

ex_ctype.spc, and ex_isxdigit.spc.

6.2.3.17 byte open (const char ∗ *mode*) [inline]

Open file.

Opens the log file. The operations that are allowed on the stream and how these are performed are defined by the mode parameter.

Parameters

mode	The file access mode. Valid values are "r" - opens the existing log file for reading, "w" - creates a new log file and opens it for writing.

Returns

The result code.

Examples:

ex_close.spc, ex_open.spc, ex_read.spc, ex_stat.spc, and ex_write.spc.

6.2.3.18 int pop (void) [inline]

Pop a value off the stack.

Pop a 32-bit integer value off the top of the stack.

Returns

The value popped off the top of the stack.

Examples:

ex_pushpop.spc.

6.2.3.19 void printf (const char * *format*, ...) [inline]

Print formatted data to debug device.

Writes to the debug device a sequence of data formatted as the format argument specifies. After the format parameter, the function expects a variable number of parameters.

Parameters

format	A constant string literal specifying the desired format.

Examples:

ex_abs.spc, ex_arrays.spc, ex_close.spc, ex_ctype.spc, ex_CurrentTick.spc, ex_DAC0ToA3.spc, ex_digiserial.-spc, ex_exitto.spc, ex_isalnum.spc, ex_isalpha.spc, ex_iscntrl.spc, ex_isdigit.spc, ex_isgraph.spc, ex_islower.-spc, ex_isprint.spc, ex_ispunct.spc, ex_isspace.spc, ex_isupper.spc, ex_isxdigit.spc, ex_open.spc, ex_printf.spc, ex_pushpop.spc, ex_read.spc, ex_rotate.spc, ex_serialout.spc, ex_sign.spc, ex_SizeOf.spc, ex_sqrt.spc, ex_-systemclock.spc, ex_timer.spc, ex_tolower.spc, ex_toupper.spc, ex_wait.spc, and ex_yield.spc.

6.2.3.20 int push (int *value*) [inline]

Push a value onto the stack.

Push a 32-bit integer value onto the top of the stack.

Parameters

value	The value you want to push onto the stack.

Returns

The value pushed onto the stack.

Examples:

ex_pushpop.spc.

6.2.3.21 char putchar (const char *ch*) [inline]

Write character to debug device.

Writes a character to the debug device. If there are no errors, the same character that has been written is returned.

Parameters

ch	The character to be written.

Returns

The character written to the file.

Examples:

ex_putchar.spc.

6.2.3.22 int puts (const char * *str*) [inline]

Write string to debug device.

Writes the string to the debug device. The null terminating character at the end of the string is not written. If there are no errors, a non-negative value is returned.

Parameters

str	The string of characters to be written.

Returns

The result code.

Examples:

ex_digiserial.spc, ex_puts.spc, ex_StartTask.spc, and t1.spc.

6.2.3.23 int read (void) [inline]

Read a value from a file.

Read a value from the file associated with the specified handle. The handle parameter must be a variable. The value parameter must be a variable. The type of the value parameter determines the number of bytes of data read.

Returns

The function call result.

Examples:

ex_close.spc, ex_open.spc, and ex_read.spc.

6.2.3.24 void RotateLeft (int & *value*) [inline]

Rotate left.

Rotate the specified variable one bit left through carry.

Parameters

value	The value to rotate left one bit.

Examples:

ex_rotate.spc.

6.2.3.25 void RotateRight (int & *value*) [inline]

Rotate right.

Rotate the specified variable one bit right through carry.

Parameters

value	The value to rotate right one bit.

Examples:

ex_rotate.spc.

6.2.3.26 void Run (const int *slot*) [inline]

Run another program.

Run the program in the specified slot. The current program will terminate.

Parameters

slot	The constant slot number for the program you want to execute. See Program slot constants.

Examples:

ex_run.spc.

6.2.3.27 char sign (int *num*) [inline]

Sign value.

Return the sign of the value argument (-1, 0, or 1). Any scalar type can be passed into this function.

Parameters

num	The numeric value for which to calculate its sign value.

Returns

-1 if the parameter is negative, 0 if the parameter is zero, or 1 if the parameter is positive.

Examples:

ex_sign.spc.

6.2.3.28 int SizeOf (variant & *value*) [inline]

Calculate the size of a variable.

Calculate the number of bytes required to store the contents of the variable passed into the function.

Parameters

value	The variable.

Returns

The number of bytes occupied by the variable.

Examples:

ex_SizeOf.spc.

6.2.3.29 int sqrt (int *x*) [inline]

Compute square root.

Computes the square root of x.

Parameters

x	integer value.

Returns

Square root of x.

Examples:

ex_sqrt.spc.

6.2.3.30 void StartTask (task *t*) [inline]

Start a task.

Start the specified task.

Parameters

t	The task to start.

Examples:

ex_StartTask.spc.

6.2.3.31 int stat (void) [inline]

Check log file status.

Check the status of the system log file.

Returns

The log file status. See Log status constants.

Examples:

ex_stat.spc.

6.2.3.32 void Stop (bool *bvalue*) [inline]

Stop the running program.

Stop the running program if bvalue is true. This will halt the program completely, so any code following this command will be ignored.

Parameters

bvalue	If this value is true the program will stop executing.

Examples:

ex_Stop.spc.

6.2.3.33 void StopAllTasks (void) [inline]

Stop all tasks.

Stop all currently running tasks. This will halt the program completely, so any code following this command will be ignored.

Examples:

ex_StopAllTasks.spc.

6.2.3.34 void StopProcesses (void) [inline]

Stop all processes.

Stop all running tasks except for the main task.

Examples:

ex_StopProcesses.spc.

6.2.3.35 int tolower (int *c*) [inline]

Convert uppercase letter to lowercase.

Converts parameter c to its lowercase equivalent if c is an uppercase letter and has a lowercase equivalent. If no such conversion is possible, the value returned is c unchanged.

Parameters

c	Uppercase letter character to be converted.

Returns

The lowercase equivalent to c, if such value exists, or c (unchanged) otherwise..

Examples:

ex_ctype.spc, and ex_tolower.spc.

6.2.3.36 int toupper (int *c*) [inline]

Convert lowercase letter to uppercase.

Converts parameter c to its uppercase equivalent if c is a lowercase letter and has an uppercase equivalent. If no such conversion is possible, the value returned is c unchanged.

Parameters

c	Lowercase letter character to be converted.

Returns

The uppercase equivalent to c, if such value exists, or c (unchanged) otherwise..

Examples:

ex_ctype.spc, and ex_toupper.spc.

6.2.3.37 void Wait (long *ms*) [inline]

Wait some milliseconds.

Make a task sleep for specified amount of time (in 1000ths of a second).

Parameters

ms	The number of milliseconds to sleep.

Examples:

ex_arrays.spc, ex_ctype.spc, ex_CurrentTick.spc, ex_DAC0ToA3.spc, ex_digiserial.spc, ex_ledcontrol.spc, ex_serialout.spc, ex_StartTask.spc, ex_systemclock.spc, ex_timer.spc, ex_Twinkle.spc, ex_TwinkleL.spc, ex_wait.spc, if_test.spc, and t1.spc.

6.2.3.38 int write (const int *value*) [inline]

Write value to file.

Write a value to the file associated with the specified handle. The handle parameter must be a variable. The value parameter must be a constant, a constant expression, or a variable. The type of the value parameter determines the number of bytes of data written.

Parameters

value	The value to write to the file.

Returns

The function call result.

Examples:

ex_stat.spc, and ex_write.spc.

6.2.3.39 void Yield (void) [inline]

Yield to another task.

Make a task yield to another concurrently running task.

Examples:

ex_yield.spc.

6.3 spmem.h File Reference

Constants defining superpro shared memory addresses.

Macros

- #define ADChannel0 @0x00
- #define ADChannel1 @0x01
- #define ADChannel2 @0x02
- #define ADChannel3 @0x03
- #define DigitalIn @0x08
- #define DigitalOut @0x09
- #define DigitalControl @0x0A
- #define StrobeControl @0x0B
- #define Timer0 @0x0C
- #define Timer1 @0x0D
- #define Timer2 @0x0E
- #define Timer3 @0x0F
- #define SerialInCount @0x10
- #define SerialInByte @0x11
- #define SerialOutCount @0x12
- #define SerialOutByte @0x13
- #define DAC0Mode @0x18
- #define DAC0Frequency @0x19
- #define DAC0Voltage @0x1A
- #define DAC1Mode @0x1B
- #define DAC1Frequency @0x1C
- #define DAC1Voltage @0x1D
- #define LEDControl @0x1E
- #define SystemClock @0x1F
- #define SharedMem01 @0x20
- #define SharedMem02 @0x21
- #define SharedMem03 @0x22
- #define SharedMem04 @0x23

- #define SharedMem05 @0x24
- #define SharedMem06 @0x25
- #define SharedMem07 @0x26
- #define SharedMem08 @0x27
- #define SharedMem09 @0x28
- #define SharedMem10 @0x29
- #define SharedMem11 @0x2a
- #define SharedMem12 @0x2b
- #define SharedMem13 @0x2c
- #define SharedMem14 @0x2d
- #define SharedMem15 @0x2e
- #define SharedMem16 @0x2f
- #define SharedMem17 @0x30
- #define SharedMem18 @0x31
- #define SharedMem19 @0x32
- #define SharedMem20 @0x33
- #define SharedMem21 @0x34
- #define SharedMem22 @0x35
- #define SharedMem23 @0x36
- #define SharedMem24 @0x37
- #define SharedMem25 @0x38
- #define SharedMem26 @0x39
- #define SharedMem27 @0x3a
- #define SharedMem28 @0x3b
- #define SharedMem29 @0x3c
- #define SharedMem30 @0x3d
- #define SharedMem31 @0x3e
- #define SharedMem32 @0x3f

6.3.1 Detailed Description

Constants defining superpro shared memory addresses. spmem.h contains declarations for superpro shared memory addresses.

License:

Author

John Hansen (bricxcc_at_comcast.net)

Date

2013-02-16

Version

2

6.3.2 Macro Definition Documentation

6.3.2.1 #define ADChannel0 @0x00

Reads the current voltage on A0 input. Value ranges from 0 to 1023. Updated every millisecond. Read only.

Examples:

ex_abort.spc, ex_arrays.spc, ex_digiserial.spc, ex_ledtest.spc, ex_printf.spc, and ex_rotate.spc.

6.3.2.2 #define ADChannel1 @0x01

Reads the current voltage on A1 input. Value ranges from 0 to 1023. Updated every millisecond. Read only.

Examples:

ex_digiserial.spc, and ex_ledtest.spc.

6.3.2.3 #define ADChannel2 @0x02

Reads the current voltage on A2 input. Value ranges from 0 to 1023. Updated every millisecond. Read only.

6.3.2.4 #define ADChannel3 @0x03

Reads the current voltage on A3 input. Value ranges from 0 to 1023. Updated every millisecond. Read only.

Examples:

ex_DAC0ToA3.spc.

6.3.2.5 #define DAC0Frequency @0x19

Control the frequency of the DAC0 analog output (O0). Read/write.

Examples:

ex_DAC0ToA3.spc.

6.3.2.6 #define DAC0Mode @0x18

Control the operation of the DAC0 analog output (O0). See Analog output mode constants for valid values. Read-/write.

Examples:

ex_DAC0ToA3.spc.

6.3.2.7 #define DAC0Voltage @0x1A

Control the voltage of the DAC0 analog output (O0). Read/write.

Examples:

ex_DAC0ToA3.spc.

6.3.2.8 #define DAC1Frequency @0x1C

Control the frequency of the DAC1 analog output (O1). Read/write.

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

6.3.2.9 #define DAC1Mode @0x1B

Control the operation of the DAC1 analog output (O1). See Analog output mode constants for valid values. Read-/write.

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

6.3.2.10 #define DAC1Voltage @0x1D

Control the voltage of the DAC1 analog output (O1). Read/write.

Examples:

ex_Twinkle.spc, and ex_TwinkleL.spc.

6.3.2.11 #define DigitalControl @0x0A

Write 8 bits to the digital control port B0 - B7. Set the mode of any of the 8 digital signals. 1 == output, 0 == input.

Examples:

ex_digiserial.spc, and ex_ledtest.spc.

6.3.2.12 #define DigitalIn @0x08

Read 8 bits from the digital port B0 - B7. Read only.

6.3.2.13 #define DigitalOut @0x09

Write 8 bits to the digital port B0 - B7. Read/Write.

Examples:

ex_digiserial.spc, and ex_ledtest.spc.

6.3.2.14 #define LEDControl @0x1E

Control the operation of the two onboard LDEs (red and blue). See LED control constants for valid values. Read-/write.

Examples:

ex_ledcontrol.spc.

6.3.2.15 #define SerialInByte @0x11

Read the next serial byte from the serial port receive queue. Reading this value removes the byte from the receive queue. Serial port input data is stored in a 255 byte temporary buffer. Read only.

Examples:

ex_digiserial.spc.

6.3.2.16 #define SerialInCount @0x10

Read the count of serial bytes in the receive queue. Enables a user program to check if any data is available to be read from the serial port. Read only.

Examples:

ex_digiserial.spc.

6.3.2.17 #define SerialOutByte @0x13

Write a byte to the serial port send queue. Serial port output data is stored in a 255 byte temporary buffer. Do not write to this address if SerialCount is 255. Write only.

Examples:

ex_serialout.spc.

6.3.2.18 #define SerialOutCount @0x12

Read the count of serial bytes in the send queue. Enables a user program to check how many bytes are waiting to be sent out the serial port. Read only.

Examples:

ex_serialout.spc.

6.3.2.19 #define SharedMem01 @0x20

Read/write via I2C at address 0x80 (msb/lsb)

6.3.2.20 #define SharedMem02 @0x21

Read/write via I2C at address 0x84 (msb/lsb)

6.3.2.21 #define SharedMem03 @0x22

Read/write via I2C at address 0x88 (msb/lsb)

6.3.2.22 #define SharedMem04 @0x23

Read/write via I2C at address 0x8C (msb/lsb)

6.3.2.23 #define SharedMem05 @0x24

Read/write via I2C at address 0x90 (msb/lsb)

6.3.2.24 #define SharedMem06 @0x25

Read/write via I2C at address 0x94 (msb/lsb)

6.3.2.25 #define SharedMem07 @0x26

Read/write via I2C at address 0x98 (msb/lsb)

6.3.2.26 #define SharedMem08 @0x27

Read/write via I2C at address 0x9C (msb/lsb)

6.3.2.27 #define SharedMem09 @0x28

Read/write via I2C at address 0xA0 (msb/lsb)

6.3.2.28 #define SharedMem10 @0x29

Read/write via I2C at address 0xA4 (msb/lsb)

6.3.2.29 #define SharedMem11 @0x2a

Read/write via I2C at address 0xA8 (msb/lsb)

6.3.2.30 #define SharedMem12 @0x2b

Read/write via I2C at address 0xAC (msb/lsb)

6.3.2.31 #define SharedMem13 @0x2c

Read/write via I2C at address 0xB0 (msb/lsb)

6.3.2.32 #define SharedMem14 @0x2d

Read/write via I2C at address 0xB4 (msb/lsb)

6.3.2.33 #define SharedMem15 @0x2e

Read/write via I2C at address 0xB8 (msb/lsb)

6.3.2.34 #define SharedMem16 @0x2f

Read/write via I2C at address 0xBC (msb/lsb)

6.3.2.35 #define SharedMem17 @0x30

Read/write via I2C at address 0xC0 (msb/lsb)

6.3.2.36 #define SharedMem18 @0x31

Read/write via I2C at address 0xC4 (msb/lsb)

6.3.2.37 #define SharedMem19 @0x32

Read/write via I2C at address 0xC8 (msb/lsb)

6.3.2.38 #define SharedMem20 @0x33

Read/write via I2C at address 0xCC (msb/lsb)

6.3.2.39 #define SharedMem21 @0x34

Read/write via I2C at address 0xD0 (msb/lsb)

6.3.2.40 #define SharedMem22 @0x35

Read/write via I2C at address 0xD4 (msb/lsb)

6.3.2.41 #define SharedMem23 @0x36

Read/write via I2C at address 0xD8 (msb/lsb)

6.3.2.42 #define SharedMem24 @0x37

Read/write via I2C at address 0xDC (msb/lsb)

6.3.2.43 #define SharedMem25 @0x38

Read/write via I2C at address 0xE0 (msb/lsb)

6.3.2.44 #define SharedMem26 @0x39

Read/write via I2C at address 0xE4 (msb/lsb)

6.3.2.45 #define SharedMem27 @0x3a

Read/write via I2C at address 0xE8 (msb/lsb)

6.3.2.46 #define SharedMem28 @0x3b

Read/write via I2C at address 0xEC (msb/lsb)

6.3.2.47 #define SharedMem29 @0x3c

Read/write via I2C at address 0xF0 (msb/lsb)

6.3.2.48 #define SharedMem30 @0x3d

Read/write via I2C at address 0xF4 (msb/lsb)

6.3.2.49 #define SharedMem31 @0x3e

Read/write via I2C at address 0xF8 (msb/lsb)

6.3.2.50 #define SharedMem32 @0x3f

Read/write via I2C at address 0xFC (msb/lsb)

6.3.2.51 #define StrobeControl @0x0B

Write 6 bits to the digital strobe port S0 - WR. Controls the operation of the six strobe outputs (S0, S1, S2, S3, RD, and WR). See Strobe control constants for valid values.

6.3.2.52 #define SystemClock @0x1F

Read the system clock. The system clock counts up continuously at one count per millisecond. Read only.

Examples:

ex_DAC0ToA3.spc, ex_systemclock.spc, ex_wait.spc, and ex_yield.spc.

6.3.2.53 #define Timer0 @0x0C

Read/write countdown timer 0. Counts down until it reaches zero (per millisecond).

Examples:

ex_abs.spc, ex_digiserial.spc, ex_ledtest.spc, ex_stat.spc, ex_Stop.spc, and ex_timer.spc.

6.3.2.54 #define Timer1 @0x0D

Read/write countdown timer 1. Counts down until it reaches zero (per millisecond).

Examples:

ex_timer.spc, and ex_write.spc.

6.3.2.55 #define Timer2 @0x0E

Read/write countdown timer 2. Counts down until it reaches zero (per millisecond).

Examples:

ex_timer.spc.

6.3.2.56 #define Timer3 @0x0F

Read/write countdown timer 3. Counts down until it reaches zero (per millisecond).

Examples:

ex_pushpop.spc, and ex_timer.spc.

7 Example Documentation

7.1 arrays.spc

This is a test of structures and arrays.

```
struct Car {
  int Make;
  int Model;
  bool TwoDoor;
};

struct Person {
  int Age;
  bool Male;
  char Initial;
  int Kids;
```

```
  Car Vehicle;
  int Weight;
  int Data[4];
};

task main()
{
  int x;
/*
  int data[5][5], buf[5], foo[2][2][2], bar[2][3][2][3];
  int *p;

  x = buf[3]; // good code

  x = data[4][3]; // bad code

  x = foo[2][3][4]; // bad code

  x++;

  Person please[4], couples[3][3]; // good allocation code
*/
  Person jch;

  jch++;
  jch.Age = 23;
  jch.Kids = 4;
  jch.Kids = jch.Kids + 2;
  jch.Initial = 'C';
  jch.Male = true;
  jch.Kids = x++;
  jch.Kids++;
  jch.Kids *= 2;
  jch.Kids--;
  jch.Vehicle.Make = 10;
  jch.Vehicle.Model = 20;
  jch.Vehicle.TwoDoor = false;
  jch.Weight = 270;
//  ++jch.Kids; // bad code
//  please[0].Age = 45; // bad code
}
```

7.2 ex_abort.spc

This is an example of how to use the abort function.

```
task main()
{
  while(true)
  {
    if (ADChannel0 > 500)
      abort(); // stop the program
  }
}
```

7.3 ex_abs.spc

This is an example of how to use the abs function.

```
task main()
{
  int x = 1000 - Timer0;
  int val = abs(x); // return the absolute value of x
  printf("x = %d, val = %d\n", x, val);
}
```

7.4 ex_arrays.spc

This is an example of how to use the Wait and printf functions.

```
task main()
{
  int i;
  int myVector[10];

  // take 10 samples spaced every 1 mS
  for (i=0; i<10; i++)
  {
    myVector[i] = ADChannel0;
    Wait(MS_1);
  }

  // print with comma separators
  for (i=0; i<10; i++)
  {
    int val = myVector[i]; // bug!!!!!
    printf("%d", val);
    printf(",");
  }
}
```

7.5 ex_close.spc

This is an example of how to use the close function.

```
task main()
{
  open("r");
  int x = read();
  printf("x = %d\n", x);
  close();
}
```

7.6 ex_ctype.spc

This is an example of how to use the ctype API functions: isupper, islower, isalpha, isdigit, isalnum, isspace, iscntrl, isprint, isgraph, ispunct, isxdigit, toupper, and tolower.

```
task main()
{
  printf("%d\n", isalnum('a'));
  printf("%d\n", isalpha('1'));
  printf("%d\n", iscntrl('g'));
  printf("%d\n", isdigit('2'));
  printf("%d\n", isgraph('%'));
  printf("%d\n", islower('G'));
  printf("%d\n", isprint('('));

  printf("%d\n", ispunct('a'));
  printf("%d\n", isspace('1'));
  printf("%d\n", isupper('g'));
  printf("%d\n", isxdigit('2'));
  printf("%d\n", tolower('.'));
  printf("%d\n", toupper('f'));

  Wait(SEC_5);
}
```

7.7 ex_CurrentTick.spc

This is an example of how to use the CurrentTick function.

```
task main()
{
  while (true) {
    Wait(MS_100);
    int x = CurrentTick();
    printf("x = %d\n", x);
  }
}
```

7.8 ex_DAC0ToA3.spc

This is an example of how to use the DAC0Mode, DAC0Frequency, DAC0Voltage, and ADChannel3 system constants as well as the Wait and printf functions.

```
task main()
{
  int A3;
  int i, c;

  // Set up Analog 0 output to output a sine wave at 1 Hz
  DAC0Mode = DAC_MODE_SINEWAVE;
  DAC0Frequency = (1);
  DAC0Voltage = 1023;

  // Sample Analog input A3 every 1/10 second
  // and output value
  while(true)
  {
    A3 = ADChannel3;
    printf("Time: %d  ", SystemClock);
    printf("A3: %d ", A3);

    // Display an '*' sine wave on terminal
    c = A3 / 20;
    for (i=0;i<c;i++) {
      printf(" ");
    }
    printf("*\n");

    Wait(MS_10);
  }
}
```

7.9 ex_digiserial.spc

This is an example of how to use the DigitalControl, DigitalOut, Timer0, ADChannel0, ADChannel1, SerialInCount, and SerialInByte system constants. It is also an example of how to use the printf, puts, and Wait functions.

```
task leds()
{
  int DigValue = 1;
  int DigCount = 8;
  int DigLen   = 7;
  int DigDir   = 0;
  DigitalControl = 0xFF;

  while(true)
  {
    DigitalOut = DigValue; // output to LEDS
    Timer0 = 10000/(ADChannel0+25);
    while(Timer0 > 0); // wait for counter to reach 0
    DigCount--;
    if (DigCount != 0)
    {
      // shifting
      if (DigDir != 0) {
        DigValue >>= 1;
      } else
        DigValue <<= 1;
      continue;
```

```
    }
    DigCount = DigLen;
    DigDir ^= 1;
    if (DigDir != 0)
    {
      // shifting
      DigValue >>= 1;
      continue;
    }

    switch(ADChannel1/128)
    {
      case 0:
        DigLen   = 7;
        DigCount = 7;
        DigValue = 0x02;
        break;
      case 1:
        DigLen   = 6;
        DigCount = 6;
        DigValue = 0x06;
        break;
      case 2:
        DigLen   = 5;
        DigCount = 5;
        DigValue = 0x0E;
        break;
      case 3:
        DigLen   = 4;
        DigCount = 4;
        DigValue = 0x1E;
        break;
      case 4:
        DigLen   = 3;
        DigCount = 3;
        DigValue = 0x3E;
        break;
      case 5:
        DigLen   = 2;
        DigCount = 2;
        DigValue = 0x7E;
        break;
      default:
        DigLen   = 1;
        DigCount = 1;
        DigValue = 0xFE;
        break;
    }

  }
}

task main()
{
  start leds;
  int b;
  while (true) {
    if (SerialInCount) {
      b = SerialInByte;
      printf("Byte %d ",b);
      if (b == 'm' || b == 'M')
        printf("Got an m or M");
      else if (b == 'H')
        printf("Got an H");
      else if (b == 'h')
        printf("Got an h");
      printf("\r\n");
    }
    else
      puts("nothing\r\n");
    Wait(1000);
  }
}
```

7.10 ex_exitto.spc

This is an example of how to use the ExitTo function.

```
// When run, this program alternates between task A and task B until halted
// by pressing the gray button.

task B();

task A()
{
   printf("task A\n");
   ExitTo(B);
}

task B()
{
   printf("task B\n");
   ExitTo(A);
}

task main()
{
   printf("task main\n");
   ExitTo(B);
}
```

7.11 ex_isalnum.spc

This is an example of how to use the isalnum function.

```
task main()
{
  printf("%d\n", isalnum('c'));
}
```

7.12 ex_isalpha.spc

This is an example of how to use the isalpha function.

```
task main()
{
  printf("%d\n", isalpha('c'));
}
```

7.13 ex_iscntrl.spc

This is an example of how to use the iscntrl function.

```
task main()
{
  printf("%d\n", iscntrl('c'));
}
```

7.14 ex_isdigit.spc

This is an example of how to use the isdigit function.

```
task main()
{
  printf("%d\n", isdigit('c'));
}
```

7.15 ex_isgraph.spc

This is an example of how to use the isgraph function.

```
task main()
{
  printf("%d\n", isgraph('c'));
}
```

7.16 ex_islower.spc

This is an example of how to use the islower function.

```
task main()
{
  printf("%d\n", islower('c'));
}
```

7.17 ex_isprint.spc

This is an example of how to use the isprint function.

```
task main()
{
  printf("%d\n", isprint('c'));
}
```

7.18 ex_ispunct.spc

This is an example of how to use the ispunct function.

```
task main()
{
  printf("%d\n", ispunct('c'));
}
```

7.19 ex_isspace.spc

This is an example of how to use the isspace function.

```
task main()
{
  printf("%d\n", isspace('c'));
}
```

7.20 ex_isupper.spc

This is an example of how to use the isupper function.

```
task main()
{
  printf("%d\n", isupper('c'));
}
```

7.21 ex_isxdigit.spc

This is an example of how to use the isxdigit function.

```
task main()
{
  printf("%d\n", isxdigit('c'));
}
```

7.22 ex_ledcontrol.spc

This is an example of how to use the LEDControl system constant as well as the Wait function.

```
task main()
{
  while(true)
  {
    LEDControl = LED_BLUE;
    Wait(SEC_1);
    LEDControl = LED_RED;
    Wait(SEC_1);
    LEDControl = LED_BLUE|LED_RED;
    Wait(SEC_1);
  }
}
```

7.23 ex_ledtest.spc

This is an example of how to use the DigitalControl, DigitalOut, Timer0, ADChannel0, and ADChannel1 system constants.

```
#pragma autostart

task main()
{
  int DigValue = 1;
  int DigCount = 8;
  int DigLen   = 7;
  int DigDir   = 0;
  DigitalControl = 0xFF;

  while(true)
  {
    DigitalOut = DigValue; // output to LEDS
    Timer0 = 10000/(ADChannel0+25);
    while(Timer0 > 0); // wait for counter to reach 0
    DigCount--;
    if (DigCount != 0)
    {
      // shifting
      if (DigDir != 0) {
        DigValue >>= 1;
      } else
        DigValue <<= 1;
      continue;
    }
    DigCount = DigLen;
    DigDir ^= 1;
    if (DigDir != 0)
    {
      // shifting
      DigValue >>= 1;
      continue;
    }

    switch(ADChannel1/128)
    {
      case 0:
        DigLen   = 7;
```

```
        DigCount = 7;
        DigValue = 0x02;
        break;
      case 1:
        DigLen   = 6;
        DigCount = 6;
        DigValue = 0x06;
        break;
      case 2:
        DigLen   = 5;
        DigCount = 5;
        DigValue = 0x0E;
        break;
      case 3:
        DigLen   = 4;
        DigCount = 4;
        DigValue = 0x1E;
        break;
      case 4:
        DigLen   = 3;
        DigCount = 3;
        DigValue = 0x3E;
        break;
      case 5:
        DigLen   = 2;
        DigCount = 2;
        DigValue = 0x7E;
        break;
      default:
        DigLen   = 1;
        DigCount = 1;
        DigValue = 0xFE;
        break;
    }

  }
}
```

7.24 ex_open.spc

This is an example of how to use the open function.

```
task main()
{
  open("r");
  int x = read();
  printf("x = %d\n", x);
  close();
}
```

7.25 ex_printf.spc

This is an example of how to use the printf function.

```
task main()
{
  int value = ADChannel0;
  printf("value = %d\n", value);
}
```

7.26 ex_pushpop.spc

This is an example of how to use the push and pop functions.

```
task main()
{
  int x = Timer3;
```

```
  int y;
  y = push(x);
  x = x*x;
  printf("x = %d\n", x);
  x = pop();
  printf("x = %d\n", x);
}
```

7.27 ex_putchar.spc

This is an example of how to use the putchar function.

```
task main()
{
  int y;
  y = putchar('\n');
  y = putchar('\t');
  y = putchar('\r');
  y = putchar('X');
}
```

7.28 ex_puts.spc

This is an example of how to use the puts function.

```
task main()
{
  puts("testing\none, two, three\n");
}
```

7.29 ex_read.spc

This is an example of how to use the read function.

```
task main()
{
  open("r");
  int x = read();
  printf("x = %d\n", x);
  close();
}
```

7.30 ex_rotate.spc

This is an example of how to use the RotateLeft and RotateRight functions.

```
task main()
{
  int x = ADChannel0;
  printf("x = %d\n", x);
  repeat(4)
    RotateLeft(x); // rotate left through carry by 1 bit (4 times)
  printf("x = %d\n", x);
  repeat(2)
    RotateRight(x); // rotate right through carry by 1 bit (2 times)
  printf("x = %d\n", x);
}
```

7.31 ex_run.spc

This is an example of how to use the Run function.

```
task main()
{
  while (true)
  {
    if (CurrentTick() > SEC_30)
      Run(SLOT2); // start running the program in slot 2
  }
}
```

7.32 ex_serialout.spc

This is an example of how to use the SerialOutCount and SerialOutByte system constants as well as the Wait and printf functions.

```
task main()
{
  while(true)
  {
    if (SerialOutCount > 200)
      continue;
    SerialOutByte = 'a';
    SerialOutByte = 'b';
    SerialOutByte = 'c';
    SerialOutByte = 'd';
    SerialOutByte = 'e';
    printf("%d\n", SerialOutCount);
    Wait(MS_500);
  }
}
```

7.33 ex_sign.spc

This is an example of how to use the sign function.

```
task main()
{
  int x = 1000 - CurrentTick();
  char val = sign(x); // return -1, 0, or 1
  printf("sign(x) = %d\n", val);
}
```

7.34 ex_SizeOf.spc

This is an example of how to use the SizeOf function.

```
task main()
{
  int x;
  bool b;
  char data[] = {1, 2, 3, 4, 5, 6};

  printf("sizeof(x) = %d\n", SizeOf(x));
  printf("sizeof(b) = %d\n", SizeOf(b));
  printf("sizeof(data) = %d\n", SizeOf(data));
  while(true);
}
```

7.35 ex_sqrt.spc

This is an example of how to use the sqrt function.

```
task main()
{
  int x = CurrentTick();
  printf("x = %d\n", x);
  x = sqrt(x);
  printf("sqrt(x) = %d\n", x);
}
```

7.36 ex_StartTask.spc

This is an example of how to use the StartTask function.

```
task foo()
{
  while(true) {
    Wait(SEC_5);
    puts("foo is running\n");
  }
}

task main()
{
  StartTask(foo); // start the foo task
  while(true)
  {
    Wait(SEC_3);
    puts("main is running\n");
  }
}
```

7.37 ex_stat.spc

This is an example of how to use the stat function.

```
task main()
{
  Timer0 = MIN_1;
  open("w");
  if (stat() == LOG_STATUS_OPEN)
  {
    write(Timer0);
  }
  close();
}
```

7.38 ex_Stop.spc

This is an example of how to use the Stop function.

```
task main()
{
  int x;
  Timer0 = SEC_10;
  while (true)
  {
    x = Timer0;
    Stop(x == 24); // stop the program if x==24
  }
}
```

7.39 ex_StopAllTasks.spc

This is an example of how to use the StopAllTasks function.

```
task main()
{
  while(true)
  {
    if (CurrentTick() > 50000)
      StopAllTasks(); // stop the program
  }
}
```

7.40 ex_StopProcesses.spc

This is an example of how to use the StopProcesses function.

```
task main()
{
  while(true)
  {
    if (CurrentTick() > SEC_30)
      StopProcesses(); // stop any tasks except for this one.
  }
}
```

7.41 ex_systemclock.spc

This is an example of how to use the SystemClock system constant as well as the Wait and printf functions.

```
task main()
{
  while(true)
  {
    printf("%d\n", SystemClock);
    Wait(SEC_5);
  }
}
```

7.42 ex_timer.spc

This is an example of how to use the Timer0, Timer1, Timer2, and Timer3 system constants. It also is an example of how to use the Wait and printf functions.

```
#define START_MS MIN_1

task main()
{
  Timer0 = START_MS;
  Timer1 = START_MS;
  Timer2 = START_MS;
  Timer3 = START_MS;
  while(true)
  {
    printf("%d\n", Timer0);
    printf("%d\n", Timer1);
    printf("%d\n", Timer2);
    printf("%d\n", Timer3);
    Wait(SEC_2);
    if (Timer2 < SEC_15)
      Timer2 = START_MS;
    if (Timer3 < SEC_30)
      Timer3 = START_MS;
  }
}
```

7.43 ex_tolower.spc

This is an example of how to use the tolower function.

```
task main()
{
  printf("tolower('C') = %d\n", tolower('C'));
}
```

7.44 ex_toupper.spc

This is an example of how to use the toupper function.

```
task main()
{
  printf("toupper('c') = %d\n", toupper('c'));
}
```

7.45 ex_Twinkle.spc

This is an example of how to use the DAC1Mode, DAC1Frequency, and DAC1Voltage system constants. It is also an example of how to use the Wait function.

```
void PlayNoteStacato(int note, int dur)
{
  DAC1Frequency = note;
  DAC1Voltage = 1023;
  Wait(7*dur/8);
  DAC1Voltage = 0;
  Wait(dur/8);
}

void PlayNoteLagato(int note, int dur)
{
  DAC1Frequency = note;
  DAC1Voltage = 1023;
  Wait(dur);
  DAC1Voltage = 0;
}

void TwinkleA()
{
    PlayNoteStacato(TONE_C3, NOTE_QUARTER);
    PlayNoteStacato(TONE_C3, NOTE_QUARTER);
    PlayNoteStacato(TONE_G3, NOTE_QUARTER);
    PlayNoteStacato(TONE_G3, NOTE_QUARTER);
    PlayNoteStacato(TONE_A3, NOTE_QUARTER);
    PlayNoteStacato(TONE_A3, NOTE_QUARTER);
    PlayNoteStacato(TONE_G3, NOTE_HALF);

    PlayNoteStacato(TONE_F3, NOTE_QUARTER);
    PlayNoteStacato(TONE_F3, NOTE_QUARTER);
    PlayNoteStacato(TONE_E3, NOTE_QUARTER);
    PlayNoteStacato(TONE_E3, NOTE_QUARTER);
    PlayNoteStacato(TONE_D3, NOTE_QUARTER);
    PlayNoteStacato(TONE_D3, NOTE_QUARTER);
    PlayNoteStacato(TONE_C3, NOTE_HALF);
}

void TwinkleB()
{
    PlayNoteStacato(TONE_G3, NOTE_QUARTER);
    PlayNoteStacato(TONE_G3, NOTE_QUARTER);
    PlayNoteStacato(TONE_F3, NOTE_QUARTER);
    PlayNoteStacato(TONE_F3, NOTE_QUARTER);
    PlayNoteStacato(TONE_E3, NOTE_QUARTER);
    PlayNoteStacato(TONE_E3, NOTE_QUARTER);
    PlayNoteStacato(TONE_D3, NOTE_HALF);
}
```

```
task main()
{
  int A0;

  // Set up Analog 1 output to output a square wave
  DAC1Voltage = 0;
  DAC1Mode = DAC_MODE_SQUAREWAVE;
  DAC1Frequency = 0;

  // Sample Analog input A0 every 1/10 second
  // and output value
  while(true)
  {
    TwinkleA();
    TwinkleB();
    TwinkleB();
    TwinkleA();
    Wait(SEC_1);
  }
}
```

7.46 ex_TwinkleL.spc

This is an example of how to use the DAC1Mode, DAC1Frequency, DAC1Voltage, and SharedMem01 system constants. It is also an example of how to use the Wait function.

```
// NXT Communications is possible with memory locations 0x20-0x3F which
// correspond to I2C addresses 0x80 to 0xff
#define NXTCOMM SharedMem01

void PlayNoteStacato(int note, int dur)
{
  if (NXTCOMM) {
    DAC1Frequency = note;
    DAC1Voltage = 1023;
    Wait(7*dur/8);
    DAC1Voltage = 0;
    Wait(dur/8);
  } else {
    DAC1Voltage = 0;
  }
}

void TwinkleA()
{
    PlayNoteStacato(TONE_C4, NOTE_QUARTER);
    PlayNoteStacato(TONE_C4, NOTE_QUARTER);
    PlayNoteStacato(TONE_G4, NOTE_QUARTER);
    PlayNoteStacato(TONE_G4, NOTE_QUARTER);
    PlayNoteStacato(TONE_A4, NOTE_QUARTER);
    PlayNoteStacato(TONE_A4, NOTE_QUARTER);
    PlayNoteStacato(TONE_G4, NOTE_HALF);

    PlayNoteStacato(TONE_F4, NOTE_QUARTER);
    PlayNoteStacato(TONE_F4, NOTE_QUARTER);
    PlayNoteStacato(TONE_E4, NOTE_QUARTER);
    PlayNoteStacato(TONE_E4, NOTE_QUARTER);
    PlayNoteStacato(TONE_D4, NOTE_QUARTER);
    PlayNoteStacato(TONE_D4, NOTE_QUARTER);
    PlayNoteStacato(TONE_C4, NOTE_HALF);
}

void TwinkleB()
{
    PlayNoteStacato(TONE_G4, NOTE_QUARTER);
    PlayNoteStacato(TONE_G4, NOTE_QUARTER);
    PlayNoteStacato(TONE_F4, NOTE_QUARTER);
    PlayNoteStacato(TONE_F4, NOTE_QUARTER);
    PlayNoteStacato(TONE_E4, NOTE_QUARTER);
    PlayNoteStacato(TONE_E4, NOTE_QUARTER);
    PlayNoteStacato(TONE_D4, NOTE_HALF);
}

task main()
{
```

```
  // Set up Analog 1 output to output a square wave
  DAC1Voltage = 0;
  DAC1Mode = DAC_MODE_SQUAREWAVE;
  DAC1Frequency = 0;

  // Play Twinkle if NXTCOMM != 0
  while(true) {
    while(NXTCOMM != 0)
    {
      TwinkleA();    if (NXTCOMM == 0) break;
      TwinkleB();    if (NXTCOMM == 0) break;
      TwinkleB();    if (NXTCOMM == 0) break;
      TwinkleA();    if (NXTCOMM == 0) break;
      Wait(SEC_1);
    }
  }
}
```

7.47 ex_wait.spc

This is an example of how to use the Wait function.

```
task main()
{
  printf("tick = %d\n", SystemClock);
  Wait(SEC_5); // wait 5 seconds
  printf("tick = %d\n", SystemClock);
}
```

7.48 ex_write.spc

This is an example of how to use the write function.

```
task main()
{
  int x = Timer1;
  open("w");
  write(x);
  close();
}
```

7.49 ex_yield.spc

This is an example of how to use the Yield function.

```
task main()
{
  printf("tick = %d\n", SystemClock);
  Yield();
  printf("tick = %d\n", SystemClock);
}
```

7.50 if_test.spc

This is a test of boolean and logic.

```
task main(){
   int x = 3;
   bool b1, b2, b3;

   if( (x=b1) && b2 && b3 && x+3>33)
      Wait( 1000 );

   x = b1 && b2 && b3;
```

```
}
```

7.51 ref_params.spc

This is an example of using reference parameter types.

```
void Bar1(int xx)
{
  xx++;
  xx += 34;
}

int Bar2(int & yy)
{
  yy++;
  yy += 34;
  return yy;
}

int Foo(int & x)
{
  Bar1(x);
  x++;
  x += 5;
  x = x*3;
//      MOV __Bar2_7qG2_yy_7qG2_000, __Foo_7qG2_x_7qG2_000 (good code)
  x += Bar2(x);
  return x;
}

task main()
{
  int y=5;
  int z;
  z = Foo(y);
}
```

7.52 t1.spc

This is an example of how to use the puts and Wait functions.

```
#pragma autostart

task t1()
{
  while(true)
  {
    puts("1 ");
    Wait(MS_500);
  }
}

task t2()
{
  while(true)
  {
    puts("2 ");
    Wait(MS_500);
  }
}

task main()
{
  start t1;
  start t2;
}
```

Index