THE SPSCL SYSTEMS PROGRAMMING LANGUAGE

For Low-level Software Implementation of Embedded Subsystems

Technical Report

Dr. José M. Garrido
Department of Computer Science

Jan, 2022

College of Computing and Software Engineering
Kennesaw State University

1 Summary

Several existing micro-controller development boards include software support for developing programs in the C and assembly languages. Examples of these types of boards are: Arduino boards and Silicon Labs Development kits. The software support provided mainly includes software tools such as: C compiler, editor, debugger, utilities for transferring the compiled program to the actual micro-controller board.

SPSCL is an experimental language was designed and its translator developed for implementing low-level programs, such as programs for micro-controllers. The language syntax is defined at a higher level of abstraction than C, and includes language statements for improving program readability, debugging, maintenance, and correctness. The language design was influenced by Ada, Pascal, C, previous generation of system programming languages such as PL/M, SPL, etc.

2 Systems Prog. Languages

The two most general classification of programming languages are: systems programming language and application programming language. A system programming language is usually used for implementing system software, such as operating systems (or parts of it), drivers, compilers, and embedded software systems. These require different development approaches when compared with application software.

System languages are designed to support high performance and access low-level facilities of the hardware. Traditionally, in addition to assembly language these languages were developed to implement operating systems on some computer systems. Examples are: SPL developed by HP for implementing the MPE operating system, ESPOL and NEWP developed by Burroughs for the MCP operating system. PL/360 and PL/S developed for the IBM S/360 operating systems. For microcomputers, PL/M was a systems programming language developed for the Intel chips.

The C programming language was originally develop in the 1970s by Dennis Ritchie at AT&T Bell Labs for implementing the Unix operating system. This language became so widely used that it also became a general-purpose programming language. Examples of more recent systems programming languages are: Rust, Go and Parasail.

The C language is the most widely used embedded programming language, with compilers available for almost every microprocessor, microcontroller, and processor core. For microcontroller software, assembly language and C are the two most commonly used programming languages.

Software development of embedded systems has become more complex baecause the trend is towards *smart* sensors and actuators. Some researchers (Stankovic) cite a need to raise the level of abstraction in software development for embedded systems. This paper partially attempts to fulfill this goal with the SPSCL systems programming language.

3 The SPSCL Language

A higher level of abstraction in design and programming improve significantly the effectiveness of software development. The SPSCL language supports the conceptual frame-work of the scientific style of computation and can facilitate the development of embedded systems, which are usually part of a large and complex system.

The language syntax is defined to be at a higher level of abstraction than C and C++ (and Java) and it is an enhancement to the widely-used pseudo-code syntax used in program and algorithm design. It includes language statements for improving program readability, debugging, maintenance, and correctness. The language design was influenced by Pascal, C, Eiffel , Ada , Java, and C++. SPSCL has retained the semantics of C.

Most developers use the compiler and tools (think debugger) provided by the chip vendor, which almost always only support C. The SPSCL language translator generates C code.

The tools typically provided by chip manufacturers are the Integrated Development Environments (IDE), a C compiler, debugger, C libraries, and sometimes an assembler. In addition to these, the SPSCL translator is also needed.

The SPSCL translator/compiler, several examples, and corresponding documentation can be downloaded for education and research purposes from the Web page:

`ksuweb.kennesaw.edu/~jgarrido/scl`

3.1 The SPSCL Translator

The SPSCL language translator is implemented as a one-pass language processor that generates C source code. The generated code can be integrated conveniently with any C or C++ library. The translator itself is an executable program implemented in C++ that has been compiled (and linked) on Linux and Windows, using the GNU C++ compiler and associated software.

The language supports the reuse of C and C++ libraries. The basic run-time support of this language is provided by the chip vendor. The translation of a SPSCL program, C compilation, and linkage are shown in Figure 1.

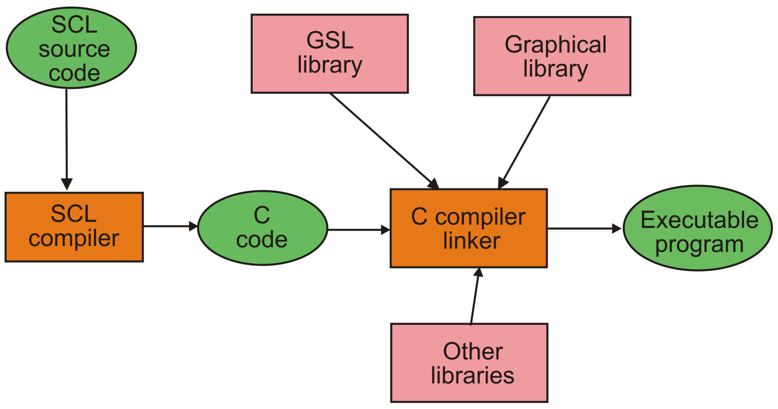

Figure 1: SPSCL Translation and C compilation/linkage

After the SPSCL translation, the generated C program needs to be compiled and linked with a C compiler before execution. CodeBlocks, CodeLite, or Eclipse can be used in developing the SPSCL programs. These tools are the most widely known IDEs for program development, and are freely available. Alternatively, a command or terminal window can be used with direct commands to translate, compile, and link SPSCL programs. For this, a good editor is necessary, such `notepad++` (on Windows) or `gedit` (on Linux).

3.2 SPSCL Features

As mentioned previously, the SPSCL language syntax is very high-level and similar to conventional pseudo-code syntax. The language statements are much easier to read, understand, and modify than the syntax of standard programming languages such as C, C++ and Java.

The general structure of an SPSCL program follows a more discipline compared to C, C++, and Java, and consists of a sequence of *sections* as follows:

1. Imports and Symbols
2. Specifications
3. Forward Declarations
4. Global Declarations
5. Implementations

3.3 Sections of the Structure of SPSCL Programs

The various sections of an SPSCL program are the following:

- **imports and symbols**, in this section one or more C header files can be included line by line. Symbolic values used in the program are be listed line by line.
- **forward declarations**, the section where declarations are written before they are completely defined.
- **specifications**, the section where the data structures (if any) are defined.
- **global declarations**, the section where global variables are defined. The data items declared are accessible by any function. These data items must be used with extreme caution because they cause difficulties in debugging and may cause synchronization problems with threads.
- **implementations**, the section where functions are completely implemented.

3.4 Relevant Language Keywords

- **import**, used to include C header files. For example:

```
import ‘‘scl.h’’
```

- **symbol**, used to list a symbolic value. For example, the following line lists symbol *MM* with a hex value (066h):

```
symbol MM 066h    // 0110 0110
```

- **define**, a keyword that declares a data item. Data is declared in three different subsections: *constants*, *variables*, and *structures*. For example:

```
constants
   define var of type byte
   define ASIZE = 50 of type integer
variables
   define marray array [] of type double
   define kelements of type integer
structures
   define abc of struct ystruct
```

- **pointer**, a keyword used to declare pointer variables or parameters. The first declaration in the following example includes a definition of a pointer variable named *palpha* of type double; the second declaration defines a pointer variable parameter, *ttime*, of struct *Timer*, to be passed by reference:

  ```
  define palpha pointer of type double
  define ttime pointer of struct Timer
  ```

- **address**, a keyword used to get the memory address of a variable. Usually, an address is assigned to a pointer variable. The following example gets the address of variable *alpha* and assigns it to pointer variable *palpha*.

  ```
   set  palpha = address alpha
  ```

- **deref**, a keyword to dereference a pointer variable. It gets the value of a variable or address pointed at by the specified pointer variable. In the following example pointer variable *palpha* is dereferenced and the corresponding value is assigned to variable *alpha* with the *set* statement:

  ```
   set  alpha = deref palpha
  ```

3.5 Overview of General SPSCL Statements

The most common language statement is the assignment statement; the keyword **set** is used as the the first symbol. The general form of this statement is:

set ⟨ *variable_name* ⟩ = ⟨ *expression* ⟩

For example,

```
set y = x + mystr.xval
```

The previous statement can also be written as follows:

```
set y = x + xval in mystr
```

When using a pointer to a structure variable, the keyword **of** is used. For example, assume *pmystr* is a pointer to the structure variable *mystr*, the previous assignment state is written:

```
set y = x + xval of pmystr
```

The bit-wise operations are **band**, **bor**, **bxor**, **negate**, **rshift**, and **lshift**. For example:

```
set varc1 = vara band varb
set varc2 = vara bor varb
set varc3 = vara bxor varb
set varc4 = negate vara
set d1 = (vara lshift 3)   // left shift 3 bits
set d2 = (varb rshift 2)   // right shift 2 bits
```

The input/output statements are **display**, **input**, and **read**. Their general forms are:

display ⟨ *data_list* ⟩
input ⟨ *string_lit* ⟩, ⟨ *var_name* ⟩
read ⟨ *var_list* ⟩

The general form of a data structure definition in SPSCL is:

```
description
      . . .

struct ⟨ struct_name ⟩ is

      data declarations (attributes)

endstruct ⟨ struct_name ⟩
```

The general form of a function is:

```
description
    . . .
  */
function ⟨ funct_name ⟩ return type ⟨ type ⟩
   parameters ⟨ param_list ⟩ is
    . . . local declarations
   begin
    . . . statements
endfun ⟨ function_name ⟩
```

The return statement is usually included in a function definition and consists of any valid expression, following the **return** keyword. The expression can include constants, variables, *struct* variables, or a combination of these.

The SPSCL statement for a *void* function call with arguments is:

```
call ⟨ function_name ⟩ using ⟨ argument_list ⟩
```

The general structure of the selection statement is:

```
if ⟨ condition ⟩
then
      ⟨ statements in Block1 ⟩
else
      ⟨ statements in Block2 ⟩
endif
```

The general structure of the *case* statement is:

```
case ⟨ selector_variable ⟩  of
   value  sel_variable_value :  ⟨ statements ⟩
     . . .
endcase
```

The general form of the while-loop is:

```
while ⟨ condition ⟩  do
      ⟨  statements in Block1 ⟩
endwhile
```

The general form of the repeat-until loop is:

```
   repeat
      ⟨ statements in Block1 ⟩
   until ⟨ condition ⟩
   endrepeat
```

The general form of the for-loop is:

```
for ⟨ counter ⟩ = ⟨ initial_value ⟩ to ⟨ final_value ⟩
   do
      Block1
endfor
```

The **file write** statement is used for output processing of an opened file for output manipulation. For example:

```
write file myoutfile "results: ", x, y
```

The **file read** statement is used for input processing of an opened file for input manipulation. For example:

```
read file myinfile z
```

4 An SPSCL Program for a Micro-Controller

The following example includes the code that demonstrates the general structure of an SPSCL program, definition of hexadecimal values, and the statements for testing the various bit-wise operations.

```
import "scl.h"
/* Program: bitops1.scl
   Test several bitwise operations
   J Garrido 3-7-2017
*/
symbol MM 066h    // 0110 0110
symbol MN 0B3h     // 1011 0011

global declarations
variables
    define a of type unsigned integer
    define b of type unsigned short
    define c of type unsigned long

implementations
  description
      This is the main function of the application.
      */
  function main is
  variables
       define vara of type byte
       define varb of type byte
       define varc1 of type byte
       define varc2 of type byte
       define varc3 of type byte
       define varc4 of type byte
       define d1 of type byte
       define d2 of type byte
  begin
       set vara = MM   // 0110 0110
       set varb = MN   // 1011 0011
       set varc1 = vara band varb
       set varc2 = vara bor varb
       set varc3 = vara bxor varb
       set varc4 = negate vara
       display varc1, " ", varc2, " ", varc3, " ", varc4
       //
       // Using a mask to select or alter bits
      set varc1 = vara band 0FEh // clear bit 0 (lowest bit)
      set varc2 = vara band 01h  // clear all except bit 0
      set varc3 = vara bor 01h   // set bit 0
      set varc4 = vara bxor 01h  // complement bit 0
      display varc1, " ", varc2, " ", varc3, " ", varc4
      set d1 = (vara lshift 3)
      set d2 = (varb rshift 2)
      display "d1: ", d1, " d2: ", d2
 //
      set a = d1         // move d1 to (low byte) of a
      display "a: ", a
      set a = a lshift 8 // shift to high byte of a
      display "a: ", a
      set a = a bor d2   // copy d2 to low byte of a
      display "a: ", a
      exit
  endfun main
```

On a command window on Windows, the SPSCL translator is invoked with the name of the SPSCL source program as the argument. The following listing shows command line interface using the translator with the SPSCL program *bitops1.scl*. This can also be translated on Linux using a Terminal window.

```
C:\SPSCL>spscl bitops1.scl
SPSCL v 1.1 File: bitops1.scl Wed Mar 11 14:28:35 2021

File: bitops1.scl no syntax errors, lines processed: 59
```

After the translation of the file *bitops1.scl*, the resulting object file *bitops1.c* can be further compiled and linked with a C compiler/linker.

The source SPSCL test programs and the generated C programs are found in the Web page mentioned previously.