C-SLang

Portable script language embedded in C code

Enhancing embedded systems testability

Version 2.0

MacroExpressions
http://www.macroexpressions.com

Table of Contents

0. C-SLANG: WHAT'S NEW IN 2.0? 3

1. C-SLANG: EXECUTIVE SUMMARY 3

1.1 WHAT IS IT FOR? 3
1.1.1 Off-board diagnostics 3
1.1.2 On-board diagnostics 4
1.1.3 Manufacturing support 4
1.1.4 Quality assurance 4
1.2 WHAT DOES C-SLANG CONSIST OF? 4
1.3 WHAT MAKES C-SLANG DIFFERENT? 4

2. ANALYSIS AND MOTIVATION 5

2.1 SOME EXAMPLES 5
2.1.1 Event-driven software 5
2.1.2 Downloadable code 5
2.2 PROBLEM ANALYSIS 5
2.2.1 Code generation issues 5
2.2.2 Data access 6
2.3 C-SLANG DESIGN GOALS 6

3. THE C-SLANG SOLUTION 6

3.1 HOST LANGUAGE 6
3.2 C-SLANG DATA TYPES AND ADDRESS SPACES 6
3.3 INSTRUCTION SET OUTLINE 8
3.4 C-SLANG VIRTUAL MACHINE 8
3.5 C-SLANG SYNTAX 9
3.5.1 Script 9
3.5.2 Functions 9
3.5.3 Function Registration 9
3.5.4 Comments and Whitespaces 9
3.5.5 Symbolic Names and Other Macros 9

4. USING C-SLANG 10

4.1 C-SLANG SCRIPT REPRESENTATIONS 10
4.1.1 Source, Bytecode and Bytecode Size in Compiled Format 10
4.1.2 Exporting a C-SLang Script Bytecode to Portable Format 10
4.2 RUNNING C-SLANG ON A VIRTUAL MACHINE 12
4.2.1 Initializing a SVIRM 12
4.2.1.1 SVIRM-related data types 12
4.2.1.2 SVIRM initialization functions 13
4.2.2 Running C-SLang scripts 13
4.3 DEBUG INTERFACE 14
4.3.1 Single-Step Execution 14
4.3.2 Breakpoints 14
4.4 COMPLETION CODES 14

5. EXAMPLES OF C-SLANG SCRIPTS 15

5.1 (SAMPLE1.C) USER MACROS 15
5.2 (SAMPLE2.C) A STRING PROBLEM 15
5.3 (SAMPLE3.C) A SIMPLIFIED SAE J1978 MESSAGE RESPONSE 16

6. INSTRUCTION SET REFERENCE 18

6.1 GENERAL 18
6.2 ADDRESS SPACES AND ARITHMETIC TYPES OF OPERANDS 19
6.3 MOVE CLASS INSTRUCTIONS 19
6.3.1 Move, MoveI *19*
6.3.2 LoadA, LoadAI, LoadX, LoadXI *20*
6.3.3 StoreA, StoreAI, StoreX, StoreXI *20*
6.3.4 MoveAX, MoveXA *21*
6.3.5 StoreAExt *21*
6.4 ARITHMETIC AND LOGIC CLASS INSTRUCTIONS 21
6.4.1 Instructions with Opcodes without endings *22*
6.4.2 Instructions with Opcodes with endings *22*
6.5 CONTROL CLASS INSTRUCTIONS 22
6.5.1 Call *23*
6.5.2 Ret *23*
6.5.3 Jump *23*
6.6 MISCELLANEOUS CLASS INSTRUCTIONS 24
6.6.1 Loop *24*
6.6.2 EndLoop *24*
6.6.3 Repeat, RepeatA, RepeatX *24*
6.6.4 CheckList *25*
6.6.5 ComputedCall and ComputedJump *26*
6.6.6 ClearA *26*
6.6.7 ClearX *26*
6.6.8 Comp1 and Comp2 ..... *27*
6.6.9 ExchangeAX *27*
6.6.10 MulDiv *27*

7. FREQUENTLY ASKED QUESTIONS 27

7.1 MY EMBEDDED APPLICATION DOES NOT USE C RUNTIME LIBRARY. CAN I STILL USE C-SLANG? 27
7.2 HOW MANY DIFFERENT SCRIPTS CAN RUN SIMULTANEOUSLY? 28
7.3 MY OPERATING ENVIRONMENT ALLOWS PRE-EMPTIVE AND COOPERATIVE TASKS. HOW DO I SCHEDULE RUNS OF C-SLANG SCRIPTS? 28
7.4 DO I NEED SEPARATE CONTROL STACKS FOR DIFFERENT SCRIPTS? 28
7.5 HOW DO I DEBUG A C-SLANG SCRIPT? 28

0. C-SLang: What's new in 2.0?

(Those unfamiliar with C-SLang can safely skip this section.)

Version 2.0 is a major rewrite. Here is the list of changes:

1. C-SLang script representation is now independent of luck with the compiler; it is conforming to C standard.
2. Accordingly, the syntax of a script header has slightly changed.
3. **`Repeat LoadA`** and **`Repeat LoadX`** operations now do useful things.
4. `StoreAExt` operation added.
5. `CallNative` operations are no longer supported since their functionality is covered by virtual physical input/output operations.
6. Integral C types of input, temporary and output variables, as well as of registers, are now supplied by the user to better match the platform. A template header is provided.
7. C-SLang virtual machine no longer owns the run control structure; it is passed as an argument. Thus, the virtual machine is reentrant and, in particular, thread-safe.
8. Virtual machine now performs runtime array boundaries check.
9. Debug interface is added via single step and breakpoint facilities.
10. Script always executes starting from the first registered function.

1. C-SLang: Executive Summary

1.1 What is it for?

C-SLang is a simple script language optimized for code density, together with its runtime environment.

The design of C-SLang is geared primarily toward the specific needs of small-size embedded systems:

- Off-board diagnostics
- On-board diagnostics
- Manufacturing support
- Quality assurance

Moreover, the small size of C-SLang scripts and very small memory footprint of the runtime environment make it suitable for writing any component where memory is at premium and execution time is available. (See some examples below.)

1.1.1 Off-board diagnostics

Once a device is released, the test procedure may still change. Example: automotive exhaust test sequence. A common solution is to *download* the test code from the test tool to the device under test and have the device execute the downloaded code.

C-SLang enables to have machine-independent code format, which is fast to download.
The test code does not change when the CPU of the device changes; this is beneficial for both the vendor of the device and for the OEM (original equipment manufacturer) or system integrator, which may be a different entity. With some standardization effort on the OEM side, the test code may even be the same for different vendors' devices.
This allows to have a stable asset library of tests, maybe different between the vendor and the OEM. Result: saving time and costs of test development and maintenance.

1.1.2 On-board diagnostics

Certain self-tests, whether power-on or continuous, are built in the device. Usually they do not require blazing speed of execution.

C-SLang allows to write those tests in a very compact and machine-independent format.
Those tests can form a test asset library.

1.1.3 Manufacturing support

There are tests that need to be run only once during the device manufacture.
For instance, when the ECU (electronic control unit) is first assembled, it is reasonable to test it for missing, crossed or shorted connections, and for basic functionality of the on- and off-chip peripherals. When an ECU is attached to the actuators (be it an antenna or a pneumatic unit), a functional test of the final assembly is in order.

C-SLang enables not to have these tests built in, but to download and execute them as needed. This reduces the code memory footprint and increases the flexibility in updating the test sets. Moreover, the functional tests, being in machine-independent format, may be independent of the particular implementation of the ECU. They may form an asset library.

1.1.4 Quality assurance

When the device fails, the problem at hand is to find the root cause of the failure and trace it to a hardware component failure, a hardware design error, software bug or manufacturing process problem. This requires explorative testing; it is not known in advance, what test pattern would identify the problem spot.

C-SLang allows to download and execute test routines that were not envisioned in advance. Moreover, the new tests, being machine-independent format, can contribute to a cumulative asset library of explorative tests.

1.2 What does C-SLang consist of?

C-SLang scripts can be used as compiled-in or as portable code. “Compiled-in” means that the script, being just a strangely looking C source code, is compiled and linked with the rest of the application. “Portable” means that the script can take a form of byte sequence, which can be executed in an application other than the one used to create it.
Thus, C-SLang comprises the following components:

- C-SLang language – a simple and compact script language
- C-SLang runtime environment, along with C-SLang API (application programmer’s interface) to initialize and run a C-SLang script
- Optional C-SLang debug API to debug C-SLang scripts (and, strictly speaking, it is a part of the runtime environment
- Optional C-SLang bytecode export facility with its own simple API.

1.3 What makes C-SLang different?

C-SLang source code is compiled into interpretable format by C compiler. Therefore, by linking script source with the interpreter, you get the compiled script automatically embedded in your application. This allows, for instance, to fully debug a C-SLang code in any luxurious integrated development environment, even if the target is a naughty microcontroller. By linking script source with the exporter, you can convert the automatically compiled script into completely portable and relocatable format.

This is what makes C-SLang unique:

1. No additional tools needed. C-SLang source code is compiled into bytecode by a standard C compiler. Therefore, C-SLang source is simply unusually looking C source. In fact, the output of the C preprocessor is a bunch of `const` data objects.
2. The main design goal was to achieve good code density. Thus, C-SLang looks rather like assembler language of a single-address machine, extended with single-instruction search, calculated call and `goto`

and some other advanced instructions. The design was inspired by architectures and instruction sets of 6800, 8086 and 8080 microprocessors, FORTRAN and Java.

3. 4-dimensional address space of the virtual machine is optimized for small components. It consists of distinct arrays of what is considered script's inputs, outputs, temporary variables, as well as (virtual) physical inputs and outputs.

2. Analysis and Motivation

In both embedded and desktop applications, there are many instances where we don't need any processing speed, but rather we want to spend as little memory as possible. Here are some generic examples.

2.1 Some Examples

2.1.1 Event-driven software

A common feature of such applications (or, rather, components) is that they comprise a system of reactions to one or few events out of many. These include human interface, some slow communications protocols, off-line diagnostic procedures etc. So, the scenario at hand is such that we have a large number of events to handle; accordingly, it requires a pretty respectable code size. At the same time, just a few events must be handled at once, so processing time is not an issue.

2.1.2 Downloadable code

In embedded applications, downloadable code is somehow loaded in the RAM of the embedded microcontroller and executed from there. Such programs can be used in software and/or hardware troubleshooting, manufacturing and quality control of the device in question. In addition, downloadable code can be shipped from one microcontroller to another, as is in case of a configurable *N: one* standby device.

In both classes of applications, execution speed is not critical, whereas saving precious RAM and ROM in embedded systems is of real importance.

2.2 Problem Analysis

A situation when "speed for size" deal is required is well known in computing community. A natural solution is to replace straight machine code with a common built-in mechanism of interpreting a (more or less) simple scripting or interpreted language. Such a scripting language should be designed to be fairly compact and easy to write in. A good example is `Tcl/Tk` for windowing systems programming.

This need seems to be well met, with interpreted languages from BASIC and Forth to Perl and Java. So, what's the need in yet another language? Well, from embedded systems prospective, there are several problems with existing languages. They are discussed below:

2.2.1 Code generation issues

Typically, interpreted code must be stored in some intermediate format, or bytecode. This is required to both save the size of representation and improve performance. (It is not true for JavaScript, for instance, – but look at the size of "interpreters"!)

To generate bytecode, we need a separate utility, whether we want to call it compiler or not. This immediately creates at least three problems:
First, it complicates project management, since this utility must be subject to archiving and version control along with the standard software toolchain.
Second, our script in source format cannot be stored in our project's normal source file; this creates problems with linking a project, especially for ROM-based systems.
Third, and for the same reason, it is extremely hard to statically embed the scripts in our application. The latter is not true, for instance, for p-code, but then p-code is bound to the application in which it is stored and cannot be exported at runtime. That's either one way or another.

2.2.2 Data access

As long as a script is written in a self-contained language, it is not a straightforward task to arrange data interface between our "normal" application and a script we want to use. Usually, a well-defined interface protocol is required, and maintaining it comes at a premium (at least of extra code size).

2.3 C-SLang Design Goals

Our design goals are to address the above-mentioned problems, in the first place. This means the following:

Translation from human-readable script to interpreter-readable bytecode must be done at compile time by means of a standard compiler. This eliminates the need in an extra tool and automatically makes the script ROMable (if needed). Anyway, if we need a script, it can be linked together with application!

To cater to the needs of downloadable code, we want to have a bytecode export capability. The logic of compiled-in script leads to a linked-in export module. Exported format should be relocatable (independent of load address) and portable (platform-independent). As an added bonus, we would be able debug scripts in any development environment we like and then export them in source code or bytecode form to the real target.
A subtle additional benefit of such an approach: there are odd microcontrollers out there, which do not execute code from certain areas of RAM. Since C-SLang bytecode is actually data, this behavior does not pose any problem.

Then, the script interpreter is split in two pieces: (a) a pretty hollow virtual machine without its own data areas and (b) a mechanism to tune it to actual data dynamically, just before running a script. One can see here an analogy with the interactions between formal and actual parameters of a subroutine. This approach solves data interface problems.

Additionally, the set of language operators and addressing modes should be chosen to reach a good compromise between complexity of the interpreter (= code size), the speed of interpretation and the size of bytecode. For best code density, the design is two-tier: common operations are optimized for code density, and actions that are more complex can be expressed *reasonably* well (not to over-inflate the interpreter).

3. The C-SLang Solution

To meet the design objectives stated earlier, C-based Script Language, or, for short, C-SLang, has been created. It is described below.

3.1 Host Language

C-SLang is implemented in portable C. One reason is that C is so popular. In fact, for some microcontrollers it may be the only high-level language available. In addition, this choice allows off-line debugging of the functionality of a given script, say, on a PC, using, for example, Microsoft VC/C++ for implementation.

3.2 C-SLang Data Types and Address Spaces

To achieve best code density, C-SLang Virtual Machine pretty closely emulates single-address machine. It can be thought of as a small processor with a pretty odd architecture.

C-SLang has two virtual registers: the Accumulator A and the Index Register X, and three (!) small-size address spaces for:

- Input variables
- Output variables
- Temporary variables

Address spaces deserve some discussion. A typical microprocessor has one or two address spaces, depending on whether I/Os are memory-mapped or have their separate space. In small tasks for which C-SLang is designed, it is usually possible to identify three kinds of memory-addressable entities – inputs, outputs and temporary storage. In

C-SLang, they are considered to be separate arrays, or address spaces. Inputs are considered read-only! It is the responsibility of the code external to C-SLang to fill in the inputs. For example, in communications application, message processing can be implemented in C-SLang. In this case, input would contain a received message, and output, - response message.

The types of registers and variables are integral C data types; they must be `typedef`'ed in the mandatory user-supplied header file "`csl_opt.h`":

- **`reg_t`** for the registers
- **`inpvar_t`** for the input variables
- **`tempvar_t`** for the temporary variables and
- **`outpvar_t`** for the output variables

A sample "`csl_opt.h`" file that comes with the C-SLang distribution has the following definitions:

```
typedef long reg_t;
typedef unsigned char inpvar_t;
typedef unsigned short tempvar_t;
typedef unsigned char outpvar_t;
```

These definitions may or may not meet your needs, and they may be changed as appropriate.
Note that if you choose an unsigned type for **`reg_t`**, the Accumulator will never be negative, so the corresponding conditional operations will never be executed.

Literals are always considered of type `unsigned char`. If you need wider numbers, you'll have to make them in the registers or variables.

While we are at it, C-SLang makes no assumption on the number of bits in a `char`; it relies on the constant `CHAR_BIT` `#define`'d in <`limits.h`>. Usually, it is 8 for a microcontroller and 16 for a DSP. In an unlikely case your compiler comes without <`limits.h`>, you can `#define` `CHAR_BIT` yourself according to your platform. Whenever a 'byte' is mentioned, the type `unsigned char` is meant.

Input, output and temporary variables are arranged in respective contiguous arrays. The maximum size of these arrays must fit a byte (e.g., 255 max if `CHAR_BIT` is 8). Indices beyond that limit could only be accessed with indexed Move instructions and, for simplicity, are not presently supported. This is not a principle limitation and can be removed if such a need is identified.

To access a variable (or a literal), you specify, roughly speaking, the type and the index, separated by comma. E.g.,
`Literal``,5` has the type `unsigned char` and the value 5.
`InpVar``,6` has the type **`inpvar_t`** and the value of whatever is stored in the element 6 of the array of input variables.

C-SLang also uses a much more "virtual" address space called virtual physical input/output (**`VPIO`**). The immediate motivation for this "type" is providing an interface to the physical inputs and outputs of the native hardware. The idea is that the application provides two functions with the following prototypes:

- `short int inp_func(unsigned ionum, reg_t *value);`
- `short int outp_func(unsigned ionum, outfunc_arg_t value);`
  where **`outfunc_arg_t`** type is `typedef`'ed in "`csl_opt.h`" by the user (and is **`reg_t`** in the default definition).

The first function reads and the "input port" by the port number and returns the value read; the second one writes a `value` to the "output port" by the port number. Either function returns a user-defined error code, 0 being OK.

Whatever the initial motivation though, notice that input and output functions can do pretty much anything they are written to do. So, this pair of functions constitutes the only interface to the target machine.

3.3 Instruction Set Outline

Complete reference of C-SLang instruction set is given in a separate section. Here we only take a glimpse at the "big picture".

C-SLang supports loading and storing data to and from registers, addition, subtraction and logical operations. It also supports copying data from one location to another bypassing the registers.

Removed from the standard set are comparisons, for their sole purpose is the following jump. Instead, C-SLang provides jumps, calls and returns conditional on the value in the Accumulator.

Jumps are considered as serious as calls. C-SLang provides (conditional and unconditional) calls and jumps to *functions*, i.e., contiguous named fragments of code. There are no jumps within a function. This makes it harder (but not impossible) to write spaghetti code.

In order to improve code density, C-SLang implements more advanced instructions, such as:

- `Minimum` – computes the lesser of the Accumulator and the operand
- `MulDiv` – multiplies *and* divides the Accumulator
- `Loop` and `EndLoop` – to arrange simple loops
- `CheckList` – provides a search for a match to the Accumulator value

Some instructions support a `Repeat` prefix, which modifies the behavior of the instruction; typically, by repeating its execution.

The only true code address arithmetic is rudimentary, yet it looks sufficient. It is computed call (or jump) where control is passed to one of several functions based on the content of the Accumulator. These instructions are reminiscent of "arithmetic IF" in FORTRAN and rely on special order of function registry in the script (see the reference below).

A variable of any data type (except literals) is considered to be an array element. In instructions supporting repeat prefix (see below) and/or indexing, repeat counter and/or index register X are considered additional offsets to the array. For instance, if the operand in an index-supporting instruction is **`TempVar,`**`5` and X=3, the operation will actually be performed on **`TempVar,`**`8` (=5+3). This provides sufficient (FORTRAN-equivalent) power of data address arithmetic.

3.4 C-SLang Virtual Machine

C-SLang virtual machine (SVIRM) is a mapping of C-SLang syntactic elements to the actual data structures. It "fleshes out" the actual arrays of input, output and temporary variables, as well as the native hardware interface (VPIO functions).

A SVIRM may be configured on the fly or statically; in any case, its definition has no reference to a C-SLang script to run. This provides maximum flexibility.

To run a script, a SVIRM runtime environment must be first initialized. It means that a data structure must be filled out to link together the C-SLang script, the SVIRM to run it on, the *control stack* to keep track of the script flow control, as well as storage for C-SLang registers and misc. data.

These concepts are discussed in detail in a section below.

It's worth making two notes here.
First, if the script was run on a runtime environment, the latter needs to be initialized again to be able to run a script, whether the same or a different one.

Second, when a C-SLang script uses VPIO operands, it obviously assumes certain functionality provided by the input and the output functions. However, these functions are supplied via a SVIRM, and so, they are separated from the script. Unless the VPIO functions are universal for the device, a mismatch is possible, and it is the programmer's responsibility to link the script with the right SVIRM. But this is also an added flexibility: by switching, e.g., the input function, you can disconnect the script from the real physical inputs of the device and, say, play back a pre-recorded sequence of inputs. This may be very helpful in tasks like failure analysis.

3.5 C-SLang Syntax

A C-SLang is a C source code; as such, it is free-input and blank characters are ignored. It is handy, however, to treat *it most of the time* as line-based: each operator takes its own line, just as a normal Assembly language. An operator can start at any position on the line.

3.5.1 Script

C-SLang script starts with the statement

`ScriptStart`(<*`script_name`*>)

and ends with the statement

`EndScript.`

There can be only one script per source file. It is possible, however, to have multiple script files.

3.5.2 Functions

The next hierarchy level is Function. Syntax:

`Function` (<*`name`*>)

The script for a function continues until the statement **`EndFunction`** is encountered. Example:

```
Function(first)
… <Operators> …
EndFunction
```

Here, the function `first` lasts until the `EndFunction` statement.
Functions do not have arguments: They all operate on common data space.

3.5.3 Function Registration

A function can only be executed if it is registered in the script. Each script starts with registration section. Syntax:

```
RegStart(
   RegFunc(<function_name>)
   ..............
   RegFunc(<function_name>)
)
```

`RegStart` must immediately follow **`StartScript`** statement.

3.5.4 Comments and Whitespaces

Since C-SLang source code is in fact a C file, C-SLang allows normal C-style comments, `/* multiline */` and `//single-line`, if the latter is supported by the C compiler.

Extra whitespaces (spaces and tabs) can be sprinkled around as desired.

3.5.5 Symbolic Names and Other Macros

Symbolic names may be `#define`'d for use instead of numbers as needed. They cannot be the same as C-SLang keywords and opcodes, of course.

Actually, C-SLang accepts C macros `#define`'ing new constructs, including function-like macros. This automatically gives some macro capabilities to C-SLang.

4. Using C-SLang

4.1 C-SLang Script Representations

4.1.1 Source, Bytecode and Bytecode Size in Compiled Format

A C-SLang script first appears in a C source file as an oddly looking fragment of code. It is automatically converted into `const` data, and this is what is called "compiled format." Notice that script in compiled format ends up in ROM if the application is ROMable. After compiling a C-SLang script, the result is a bunch of initialized constants in constants' area (be it section, segment or group, depending on the platform).

Note that to compile a script, the header files `csldef.h` and `cslpubl.h` must be `#include`'d.

The size of the script can be calculated as follows:

```
  3*sizeof(char const *)  per script
+ sizeof(char const *) * number-of-registered-functions
+ the sum of lengths of all functions
```

The length of a function is the sum of lengths of all its instructions plus 1, rounded up to the next alignment boundary for `char const` array.

Example: consider a trivial script

```
ScriptStart(GoofyExample)
RegStart(
      RegFunc(DoNothing)
)
Function(DoNothing)
EndFunction
EndScript
```

Assuming all pointers are four bytes long, this script takes 3*4 per script + 4*1 for registered functions + 1 byte for an empty function, totaling 17 bytes.

Note1: If your compiler pads const character arrays to an alignment boundary, the length of a function, for the purpose of this calculation, should be rounded up to the nearest boundary. The example above would then give 20 bytes.

Note 2: Some compilers treat a C construct used in C-SLang script as suspicious; they issue a warning. Some compilers may even generate an error. In either case, you can work around this by inserting a line

```
#define CSL_CONST
```

before including C-SLang headers. The price for that is an intermediate pointer of the script becomes non-const, so a script would carry a pointer-size RAM overhead.

4.1.2 Exporting a C-SLang Script Bytecode to Portable Format

A C-SLang script in compiled format is automatically generated and is ROMable all right, but it is neither portable nor relocatable. This is not because of C-SLang language itself but because of limitations of C pre-processor. To convert compiled format into portable format, use the function `CSLangBytecodeExport`. Here are the prototypes:

```
typedef void ByteDump_t(int c, void *arg);
extern int CSLangBytecodeExport(CSLscript_t script, ByteDump_t *emitFunc,
                                void *arg);
```

`CSLangBytecodeExport` takes three arguments: a **`CSLscript_t`** C-SLang script in compiled format, a pointer to a *byte export* function and a pointer argument passed transparently the bytecode export function.

A bytecode export function is a user-supplied void function that takes an `int` argument – a byte of the script in export format, and a `void*` pointer that is interpreted as appropriate for the particular function.
`CSLangBytecodeExport` calls the byte export function for each byte produced, and it can do with those bytes whatever it is designed for, e.g.:

- To collect as an array of bytes,
- To save in a disk file,
- To send over network,
- Alternatively, just to collect statistics about the script.

`CSLangBytecodeExport` calls the emit function repeatedly for every consecutive byte of the exported script. The bytes are emitted in the following order

- Bytecode length (in bytes; including the length bytes), high byte
- Bytecode length, low byte
- The number of functions in the script
- Function registry, multiple bytes
- Bytecode of the registered function, multiple bytes

For example, exporting a trivial script from the previous section, the following bytes would be emitted:
0, 6 (length), 1 (number of functions), 0, 5 (offset of the function), 0Xxx (function termination symbol, the only functional byte in the bytecode): total of 6 bytes.

For an example, let's make a simple function calculating the length of the script in exported format:

```
struct length {unsigned char len[2]; unsigned short byte_count};
void len_dump(int c, void *arg)
{
    struct len_dump *p = arg;
    if(p->byte_count < 2) p->len[p->byte_count++] = (unsigned char)c;
    /* do nothing after collecting the two length bytes */
}

int script_length(CSLscript_t script)
{
     struct length L;
     int ret;
     L.byte_count = 0;
     ret = CSLangBytecodeExport(script, len_dump, &L);
     if(ret < 0) return ret; /* error code */
     return (L.len[0]<<CHAR_BIT)|L.len[1];
}
```

For debugging purposes one can use an emit function as simple as

```
      unsigned char myBytecode[200];
      int mycount=0;

      void mydump(int c)
      {
            myBytecode[mycount++] = (unsigned char)c;
      }
```

This simply fills character array with emitted bytecode. Real world emit function(s) can be, of course, more sophisticated. To create download modules written in C-SLang, emit function might write its argument character to the output file on disk.

4.2 Running C-SLang on a Virtual Machine

There can be several C-SLang virtual machines (SVIRMs) in an application; you can choose, for instance, to run one script on different SVIRMs or different scripts on a single SVIRM. To pick a SVIRM, we define it as a variable of type **`SVIRM_t`**. (To demystify it: **`SVIRM_t`** is simply a `struct` type `typedef`'ed in the header `cslpubl.h`. So, if you find instruction by example inadequate, you can see all the fields of the structure in the header.)

4.2.1 Initializing a SVIRM

4.2.1.1 SVIRM-related data types

To run a C-SLang script, a C-SLang virtual machine, SVIRM, must be first *initialized*. This means, it must be told what "personality" to assume, that is, what are its

1. Input array
2. Output array
3. Temporary array
4. Input read function
5. Output write function

This information is aggregated in the following data type (from the header file `cslpubl.h`):

```
/* -- C-SLang Virtual Machine data type -- */
typedef struct SVIRM_t{
      inpbuf_t inp;
      outpbuf_t outp;
      tempbuf_t tempbuf;
      InpFunc_t *infunc;
      OutpFunc_t *outfunc;
} SVIRM_t;
```

where the **`InpFunc_t`** and **`OutpFunc_t`** are the types of "virtual physical" input and output functions respectively:

```
typedef reg_t InpFunc_t(unsigned ionum);
typedef short int OutpFunc_t(unsigned ionum, outfunc_arg_t value);
```

and **`inpbuf_t`**, **`outpbuf_t`**, **`tempbuf_t`**, are the types of arrays of inputs, outputs and temporaries respectively, specified by the start pointer and the length:

```
typedef struct {
      const inpvar_t *buf;
      unsigned char size;
} inpbuf_t;

typedef struct {
      outpvar_t *buf;
      unsigned char size;
} outpbuf_t;

typedef struct {
      tempvar_t *buf;
```

```
      unsigned char size;
} tempbuf_t;
```

SVIRM_t structures can be kept `const` if so desired.

Additionally, the SVIRM will use two RAM areas while running the script.
One contains current run information – emulated registers, status and so on; it has fixed size, and it is generally of no interest to the user. It must be instantiated though in the user code, like

```
CSLangRun_t myrun;
```

so, its type, **CSLangRun_t**, is provided in a header automatically included with `slpubl.h`.

The second RAM data element is a so-called control stack; it is used to keep track of C-SLang constructs that change natural program flow of the script (e.g., function calls). The right size of control stack is anyone's guess, just as size of normal stack in normal programming. If you know in advance what scripts you are going to run, you can choose the size at compile time. One way or another, the user must define an array of type **CtlStack_t** (`#define`'d in `slpubl.h`) to the SVIRM, something like

```
CtlStack_t ctlstack[MAX_DEPTH];
```

and supply it in your program to the SVIRM.

4.2.1.2 SVIRM initialization functions

To initialize a SVIRM, you call one of the two functions, depending on whether the script to run is in compiled or in exported format. As always, freedom (of configuration) comes at a price. Initialization of SVIRM is something that very few people will call elegant.

If the `script` is in compiled format, the initialization function prototype is this:

```
int CSLangInitCom(CSLangRun_t *run, CSLscript_t script,
            SVIRM_t const *svirm, CtlStack_t *stack, unsigned short n);
```

Here, `run` is the pointer to runtime environment to be initialized, script is the script to run, `svirm` is the memory and I/O configuration, `stack` is the pointer to a control stack array and `n` is the number of elements in the control stack array.
The function returns an error code, 0 being "OK." Error codes are described in a section further below.

If the `script` is in exported format, the initialization function prototype is this:

```
int CSLangInitExp(CSLangRun_t *run, const unsigned char *script,
            SVIRM_t const *svirm, CtlStack_t *stack, unsigned short n);
```

The only difference is the type of the `script` argument, which in exported format is a sequence of bytes.

This interface is flexible enough to allow reusing memory areas via mix-and-match: One can initialize one or several run structures with different scripts or the same script, and using the same or different SVIRMs or control stacks.
In particular, combining a script with different SVIRMs can produce rather cool effects. For instance, to disconnect the C-SLang engine from physical inputs of the system and have it use pre-recorded inputs played back, you just replace the SVIRM with another SVIRM with accordingly different input function.

4.2.2 Running C-SLang scripts

To run a C-SLang script, after runtime environment has been initialized, you just call the function **CSLangExec**; its prototype is

```
int CSLangExec(CSLangRun_t *run);
```

The function returns a completion code, `CSL_DONE` indicating the successful end of the script execution.

4.3 Debug Interface

If C-SLang engine is compiled with `CSL_DEBUG` `#define`'d in the header file `csl_opt.h` (default) then a limited debug interface is available. It can be useful for debugging C-SLang scripts, since the programmer can monitor the script execution progress and inspect public members of the **`CSLangRun_t`** structure (such as the virtual registers)

4.3.1 Single-Step Execution

After runtime environment has been initialized, you can call **`SCLangSetSingleStep`** to enable or disable (default) single-step execution. The function's prototype is

```
void SCLangSetSingleStep(CSLangRun_t *run, unsigned char sstep);
```

If `sstep` is a zero, single step is disabled (default); otherwise, it is enabled.

If single step is enabled, **`CSLangExec`** returns after executing one C-SLang instruction; if there were no errors, it returns `CSL_DONE` if the end of the script was reached, or `CSL_STEP_COMPLETED` otherwise. To continue script execution after **`CSLangExec`** returns `CSL_STEP_COMPLETED`, call it again. Some sort of a loop is a likely arrangement here.

Single step can be enabled or disabled at any time.

4.3.2 Breakpoints

A breakpoint specifies the function (by 0-based number of it in the script registry section) and the (0-based) instruction number within the function. Breakpoints are arranged as a list with last element referencing `NULL`:

```
typedef struct CSLbp_t{
      struct CSLbp_t *next;
      unsigned char func_id;
      unsigned short instr;
} CSLbp_t;
```

To set breakpoints – the whole list at once – call the function **`SCLangSetBreakpoints`**; its prototype is here:

```
void SCLangSetBreakpoints(CSLangRun_t *run, const CSLbp_t *breakpoints);
```

To change the set of breakpoints, call **`SCLangSetBreakpoints`** with another list. To disable breakpoints, pass `NULL` as the list pointer.

If single step is enabled, breakpoints have no effect. Otherwise, **`CSLangExec`** will return `CSL_BREAKPOINT` each time it successfully reaches the point *just before* executing an instruction identified in one of the breakpoints in the list.

Similar to single-step, to continue script execution after **`CSLangExec`** returns `CSL_BREAKPOINT`, call it again. Some sort of a loop is a likely arrangement here, too.

4.4 Completion Codes

On success, **`CSLangExec`** returns one of the following *positive* codes, `#define`'d in `cslpubl.h`:

- `CSL_DONE` – Run completed; no errors encountered
- `CSL_STEP_COMPLETED` – An instruction completed in single-step mode
- `CSL_BREAKPOINT` – A breakpoint is reached

`CSLangExec`, **`CSLangBytecodeExport`**, **`CSLangInitCom`** and **`CSLangInitExp`** may detect and return the following *negative* codes, also `#define`'d in `cslpubl.h`:

- `CSL_CALLS_TOO_NESTED` – control stack turned out to be of insufficient size

- `CSL_UNIMPLEMENTED_OPCODE` – the instruction requested is not implemented. If your script is generated correctly, you should never see this code
- `CSL_NONWRITEABLE_TARGET` – an attempt is made to move or store data in a **`Literal`** or **`InpVar`**
- `CSL_WILD_EOLOOP` – an **`EndLoop`** instruction was encountered without a matching **`Loop`** instruction
- `CSL_ILLEGAL_REPEAT` – a **`Repeat`** prefix was used with an instruction that does not admit it
- `CSL_BADINSTR` – a bad instruction in the script. If your script is generated correctly, you should never see this code
- `CSL_UNIMPLEMENTED_FUNCTION` – a **`VPIO`** (input or output) function call was required but the NULL function was specified in the SVIRM
- `CSL_INDEX_LIMIT` – array limit exceeded. The same code is used for input, output and temporary array boundary violation
- `CSL_ZERODIVIDE` – an attempt to divide by 0
- `CSL_FUNCTION_INDEX` – an invalid destination function index for a script function
- `CSL_CSRIPT_TOO_LONG` – script length doesn't fit 2 bytes
- `CSL_BADARG` – an invalid argument was passed, such as NULL runtime environment

In addition, **`CSLangExec`** considers a non-zero value returned by a function as an error (such as a manifestation of an I/O error). If **`CSLangExec`** encounters such a value, it is treated as the error code; **`CSLangExec`** immediately returns to the caller with the propagated error code.

Here lies a great opportunity to confuse C-SLang engine by writing VPIO functions that return codes conflicting with those reserved by C-SLang. Don't do that! It is a good practice to have VPIO functions return *negative* error codes less than –512.

5. Examples of C-SLang Scripts

C-SLang distribution comes with a few sample files. One purpose of them is to provide a small regression test: the test application linked with them should give the same results as the reference results file provided in the distribution. Another purpose of the sample files is to demonstrate the use of C-SLang.

Here is a brief description of the sample files.

5.1 (sample1.c) User macros

This file illustrates the fact that you can #define C-SLang macros like C macros, and use them as an extension of the C-SLang instruction set. E.g.,

```
#define div256 \
    MulDiv(Literal, 1, 2) \
    MulDiv(Literal, 1, 128)
. . . . . .
    Repeat(Literal, 4) LoadA(InpVar, 0)
    div256 /* illustrates a macro */
    Repeat(Literal, 4) StoreAExt(OutpVar, 0) /*store divided value*/
```

This fragment also illustrates a use of the **`Repeat`** prefix with **`LoadA`** and **`StoreAExt`** instructions.

5.2 (sample2.c) A string problem

This is an example of a string problem for which class C-SLang was *not* optimized: to find the number of occurrences of the first character in the input string. Here is the complete code:

```
ScriptStart (Charcount)
RegStart(
   RegFunc(charcount)
   RegFunc(charhelper)
```

```
   RegFunc(increment)
)
/*Count the number of occurences of the first
  character in the input string
*/
Function (charcount)
    ClearX
    StoreX(TempVar, 1) /*counter*/
    Call(charhelper) /*a hand-made loop must be a separate function */
    Move(TempVar, 1, VPIO, 0) /*output the result*/
EndFunction
/*Illustrates a function wrapper for a loop*/
Function (charhelper)
    LoadAI(InpVar, 1) /* InpVar[1+X] loaded */
    Subtract(Literal, 0) /*Check for 0 character*/
    RetZ                 /*return if end*/
    Subtract(InpVar, 0) /*Compare with char*/
    CallZ(increment)     /*Increment if equal*/
    MoveXA               /*Increment index X */
    Add(Literal, 1)
    MoveAX
    Jump(charhelper) /* repeat */
EndFunction
Function (increment)
    LoadA(TempVar, 1) /* increment TempVar 1 */
    Add(Literal, 1)
    StoreA(TempVar, 1)
EndFunction
EndScript
```

This simple script illustrates a technique to arrange a loop (see **`Function`** (`charhelper`)). Since we don't know the string length in advance, we cannot use the **`Loop/EndLoop`** mechanism. On the other hand, **`Jump`**s are allowed only to functions. So, we have to extract the loop body into a separate function. Note that if the input array does not contain a 0 character, the execution of this script would continue until the array boundary is hit, at which point the execution would terminate with error code `CSL_INDEX_LIMIT`.

The call of `increment` within `charhelper` also illustrates that a conditional block must be a separate function.

Finally, this script illustrates that counting via addition is rather awkward, and a counting instruction is probably the most likely to be implemented in the next release of C-SLang.

5.3 (sample3.c) A simplified SAE J1978 message response

This section describes a simplified yet more involved example. The example comes from serial communications message processing area and is related to SAE J1978 diagnostic messages over SAE J1850 network with 3-byte header format. If this description looks intimidating, fear it not. We simply assume that **`InpVar`** contains a received message (**`InpVar`** 0 being its length), and we must process it as follows:

1. If the length less than 4, ignore the message
2. If the five least significant bits of the first byte of the message are wrong, ignore it
3. If byte 2 is not 0x10 or 0xFE, ignore the message
4. If bit 6 of byte 4 is set, ignore the message
5. If byte 4 ("mode") is found among "supported modes" 0x12, 0x14, 0x17, 0x19, 0x20 then prepare a "positive response" in the **`OutpVar`** array and store the "mode" in **`VPIO`** 0 (to cause the appropriate native processing); otherwise, prepare a "negative response" in **`OutpVar`**. Again, byte 0 of the **`OutpVar`** is the length of the response message.

6. For simplicity, our positive response will be 4 bytes long; for an input message
   Byte1, Byte2, Byte3, Byte4, …
   it will be
   Byte1, Byte3, 0x10, (Byte4 | 0x40).
7. Our negative response will be
   Byte1, Byte3, Byte2, 0x7F, Byte4, …, ByteLast, 0x11,
   i.e., all received message bytes from Byte4 to the last byte (but no more than Byte9) are copied to the **OutpVar**, followed by error code 0x11.

Here is an implementation:

```
ScriptStart (SAE_example)
RegStart(
   RegFunc(SAEMessage)
   RegFunc(PositiveResp)
   RegFunc(NegativeResp)
   /* mode-specific functions are registered
      in the order convenient for ComputedCall
   */
   RegFunc(Mode12)
   RegFunc(Mode14)
   RegFunc(Mode17)
   RegFunc(Mode19)
   RegFunc(Mode20)
)
Function( SAEMessage )
   Move(Literal, 0, OutpVar, 0) /* resp. length 0 means 'ignore' */
   LoadA(InpVar, 0) /* length */
   Subtract(Literal, 4)
   RetNeg   /* if too short a message */
   LoadA (InpVar, 1)
   /* check header format */
   And(Literal, 0x1F)
   Subtract(Literal, 0x0C)
   RetNotZ  /*return if wrong format*/
   /* check the target address */
   CheckList(InpVar, 2, 2, Literal, 0x10 CSLIST 0xFE)
   RetNeg /*return if no match*/
   /* check if it is a request */
   LoadA(InpVar, 4)
   And(Literal, 0x40)
   RetNotZ /* return if not a request */
   /* check if the mode is supported */
   CheckList(InpVar, 4, 5, Literal,
        0x12 CSLIST
        0x14 CSLIST 0x17 CSLIST
        0x19 CSLIST 0x20)
   JumpNeg(NegativeResp) /* not supported: negative response */
      ComputedCall(Mode12)  /* mode supported: process it */
      Call(PositiveResp)    /* and prepare positive response */
EndFunction

Function (PositiveResp)
      Move(Literal, 4, OutpVar, 0) /*default response length*/
      Move(InpVar, 1, OutpVar, 1)  /* first three bytes are header */
      Move(InpVar, 3, OutpVar, 2)
```

```
      Move(Literal, 0x10, OutpVar, 3)
      LoadA(InpVar, 4)
      Or(Literal, 0x40) /*mode: response to the request*/
      StoreA(OutpVar, 4)
EndFunction

Function(NegativeResp)
      Call (PositiveResp) /* make header */
      Move(Literal, 0x7F, OutpVar, 4) /*negative response*/
      LoadA( InpVar, 0 )            /*Length of the input message*/
      Add(Literal, 2)
      Minimum(Literal, 11)
      StoreA(OutpVar, 0)      /*length of neg. response */
      MoveAX
      MoveI(Literal, 0x11, OutpVar, 0) /* response code => the last byte */
      Subtract(Literal, 5)    /* - Header length */
      RetNegZ                /* return if nothing to copy*/
      RepeatA     /* Is the number of bytes to copy */
            Move(InpVar,4, OutpVar,5)
EndFunction
/* dummy mode processing functions */
Function( Mode12 )
      Move(Literal, 0x12, VPIO, 0)
EndFunction
Function( Mode14 )
      Move(Literal, 0x14, VPIO, 0)
EndFunction
Function( Mode17 )
      Move(Literal, 0x17, VPIO, 0)
EndFunction
Function( Mode19 )
      Move(Literal, 0x19, VPIO, 0)
EndFunction
Function( Mode20 )
      Move(Literal, 0x20, VPIO, 0)
EndFunction
EndScript
```

This script illustrates the use of the **`CheckList`** instruction in `SAEMessage`. It is used one time to check the message target address match, and the second time to arrange computed call to a request-specific processing function.
The end of `NegativeResp` demonstrates array copying with a single prefixed instruction.

6. Instruction Set Reference

6.1 General

A function in C-SLang is a sequence of instructions. Operands of an instruction depend on the instruction itself. Generally, they are of the variable spaces described earlier. Some instructions restrict the types of operands.

The instruction set was designed with resulting code density as the highest priority. The resulting language bears some resemblance of Motorola 6800 and Intel 8086 Assemblers as well as FORTRAN. A deeper look might also find remnants of COBOL and Java in this eclectic mix. The result is not a truly elegant language but it does provide a very good code density.

All instructions fall into one of the four categories:

- Move
- Arithmetic (and logic)
- Control
- Miscellaneous

These categories are described in detail below.

6.2 Address Spaces and Arithmetic Types of Operands

A numeric operand of a C-SLang instruction must have a type attribute, which identifies its address space. The following attributes are valid:

- **`Literal`** – an unsigned byte-wide value, `unsigned char`
- **`InpVar`** – an element of the array of "input variables," **`inpvar_t`**
- **`OutpVar`** – an element of the array of "output variables," **`outpvar_t`**
- **`TempVar`** – an element of the array of "temporary variables," **`tempvar_t`**
- **`VPIO`** – if a type of a source operand, a value produced by the "input function," **`reg_t`**; if a type of a destination operand, a value used by the "output function," **`outfunc_arg_t`**.

It is important to understand how the user-supplied types affect the results of C-SLang instructions.

For uniformity, and in order to circumvent "undefined behavior" of C constructs, the value of a source operand is first promoted to **`reg_t`** and then, for all instructions *except those including comparison (**`Minimum`** and **`CheckList`**)*, to `unsigned long`. It means that on a typical binary machine, if the type of source operand is narrower than `unsigned long`, then it is zero-extended if the operand type is unsigned, and it is sign-extended if the operand type is signed.

The operation is performed on `unsigned long` operand(s) and the result is demoted to the type of the destination operand. On a typical binary machine, it means discarding the higher-order bits that do not fit the size of the destination operand.

This logic may be simplified or entirely optimized out by the compiler, but it is necessary to understand the semantics of it.

With respect to arithmetic operations, this logic means that multiplication and division work on unsigned types and that addition and subtraction would work with signed types as expected on machines where negative numbers are represented in 2's complement format (which is nowadays typical if not universal).

It is recommended that **`tempvar_t`** be at least as wide as **`inpvar_t`** and **`outpvar_t`**, and that **`reg_t`** be at least as wide as **`inpvar_t`**, **`outpvar_t`** and **`tempvar_t`**.

6.3 Move Class Instructions

As the class name implies, these instructions move (in fact, copy) data from one location to another. If a quantity is moved to a wider location, it is extended according to its type: Unsigned types are zero-extended and signed types are sign-extended. If a quantity is moved to a narrower location, only the corresponding number of its least-significant bits is actually moved.

Important note: All Move class instructions can be modified by `Repeat` prefixes. See description of **`Repeat`** for details.

6.3.1 Move, MoveI

General-purpose move (**`Move`**) or indexed Move (**`MoveI`**)
Syntax:

`Move`(*`<source_type>, <source_number>, <target_type>, <target_number>`*)

The instruction moves parameter number <*source_number*> of the type <*source_type*> to parameter number <*target_number*> of the type <*target_type*>.

MoveI(<*source_type*>, <*source_number*>, <*target_type*>, <*target_number*>)
The instruction moves parameter number <*source_number*>+IndexRegister of the type <*source_type*> to parameter number <*target_number*>+ IndexRegister of the type <*target_type*>. If <*source_type*> is **Literal**, though, the value is not indexed.

Examples:

Move(**Literal**, 17, **OutpVar**, 4)
This assigns a value of 17 to the output variable 4.
MoveI(**Literal**, 17, **OutpVar**, 4)
Assuming index register X=6, this assigns a value of 17 to the output variable 4+6=10.
Move(**VPIO**, 0, **VPIO**, 1)
The input function is called with the argument 0; then the output function is called with the value returned by the input function as the first argument, and with 1 as the second argument.

Length:

3 bytes.

Restrictions:

<*target_type*> cannot be **Literal** or **InpVar**.

6.3.2 LoadA, LoadAI, LoadX, LoadXI

Load pseudo-registers (Accumulator A or index register X).

Syntax:

Load<***Reg***>[**I**](<*source_type*>, <*source_number*>)
where <*Reg*> is either **A** or **X**. The instruction moves parameter number <*source_number*> (or, if the '**I**' suffix is supplied, <*source_number*>+(content of X)) of the type <*source_type*> to the named register.
No indexing is performed on literals.
IMPORTANT: See Repeat description on how Load instructions are affected by it.

Example:

LoadA(**Literal**, 17)
This assigns a value of 17 to the accumulator.

Assuming index register X = 6,
LoadXI(**Literal**, 17)
assigns a value of 17 to X. Using **LoadXI** with literals makes code more obscure. Use **LoadX** instead.
LoadXI(**TempVar**, 17)
This assigns a (possibly, promoted) value of temporary variable 23(=17+6) to the index register.

Length:

2 bytes.

Restrictions:

None.

6.3.3 StoreA, StoreAI, StoreX, StoreXI

Stores accumulator A or index register X.

Syntax:

Store<***Reg***>[**I**](<*target_type*>, <*target_number*>)
where <*Reg*> is either **A** or **X**.
The instruction copies the value in the named register to the parameter number <*source_number*> (or, if '**I**' suffix is supplied, <*source_number*>+(content of **X**)) of the type <*source_type*>.

Examples:

StoreA(**OutpVar**, 17)
This assigns a value from accumulator to output variable 17.
Assuming index register X = 6,
StoreXI(**TempVar**, 17)
This assigns the value 6(=**X**) to temporary variable 23(=17+6).

Length:
2 bytes.

Restrictions:
<*target_type*> cannot be **Literal** or **InpVar**.

6.3.4 MoveAX, MoveXA

Moves accumulator **A** to index register **X** (**MoveAX**), or **X** to **A** (**MoveXA**).

Syntax:
MoveAX
MoveXA

Example:
Assuming index register X = 6, accumulator A=1234
MoveAX
This assigns the value 1234(=**A**) to **X**.
Assuming again index register X = 6, accumulator A=1234
MoveXA
This assigns the value 6(=**X**) to **A**.

Length:
1 byte.

Restrictions:
None.

6.3.5 StoreAExt

Stores the least-significant byte (CHAR_BIT bits) of the Accumulator in the destination address and shifts the Accumulator a byte to the right. The CHAR_BIT most-significant bits of the Accumulator are filled with zeros regardless of whether the **reg_t** type is signed or not.

Syntax:
StoreAExt<*target_type*>, <*target_number*>)

Examples:
StoreA(**OutpVar**, 17)
Assuming A=0xFEDCBA55 and CHAR_BIT=8, the instruction writes 0x55 in the output variable 17 and 0x00FEDCBA to the Accumulator.

Length:
2 bytes.

Restrictions:
<*target_type*> cannot be **Literal** or **InpVar**.

6.4 Arithmetic and Logic Class Instructions

All arithmetic and logic instructions perform an operation on the accumulator A and an operand and store the result back in the accumulator. The opcodes can have optional ending A or X; in this case, the operand is the Accumulator itself or the Index Register, respectively.

The following instructions are supported:

- **Add** – addition
- **Subtract** – subtraction

- **`Or`** – bitwise logical OR
- **`And`** – bitwise logical AND
- **`XOr`** – bitwise logical exclusive OR
- **`Minimum`** – minimum
- **`MulDiv`** – multiplication and division

6.4.1 Instructions with Opcodes without endings

These instructions perform a binary operation on the accumulator **`A`**.
Syntax (generic):

`Opcode(<source_type>, <source_number>)`
The instruction performs the `Opcode` operation on the accumulator register and parameter number <*`source_number`*> of the type <*`source_type`*> and stores the result in the accumulator register.

Example:

`Add(Literal``, 17)`
This adds 17 to the accumulator.

Length:

2 bytes.

Restrictions:

None.

6.4.2 Instructions with Opcodes with endings

These instructions perform a binary operation on the accumulator **`A`**.
Syntax (generic; ending E stands for **`X`** or **`A`**):

`Opcode<E>`
The instruction performs `Opcode` operation on the accumulator register and (depending on the ending being **`A`** or **`X`**) the accumulator or the index register and stores the result in the accumulator register.

Example:

`AddA`
This doubles the accumulator.
`SubtractX`
This subtracts the index register from the accumulator.

Length:

1 byte.

Restrictions:

Redundant or useless opcodes, like `XOrA` are not defined or implemented.

6.5 Control Class Instructions

Instructions in this class change the flow control of the script execution. The following instructions are supported:

- `Call`
- `Ret`
- `Jump`

The first three instructions change flow control within the same script. The last one calls (by number) a function listed in native function table.

Each of the four instructions can have an ending making them execute conditionally depending on the (signed) value of accumulator register A. The endings are:

- `Z` – execute if `A==0`; otherwise skip to the next instruction
- `NotZ` – execute if `A!=0`; otherwise skip to the next instruction

- `Pos` – execute if `A>0`; otherwise skip to the next instruction
- `PosZ` – execute if `A>=0`; otherwise skip to the next instruction
- `Neg` – execute if `A<0`; otherwise skip to the next instruction
- `NegZ`, – execute if `A<=0`; otherwise skip to the next instruction

Note: `Ret` **must** have an ending.
Note: If the user-supplied register type, **`reg_t`**, is an unsigned type, `Neg` is useless and `PosZ` is unconditional.

6.5.1 Call

Syntax:

`Call``[`*`Ending`*`](<`*`Function_name`*`>)`

This instruction passes the control to the function <*`Function_name`*>. After the called function returns, control is passed to the instruction following the **`Call`**.

Example:

`CallNotZ``(Police)`

This executes the function `Police` in the same script, but only if accumulator is non-zero.

Length:

2 bytes

Restrictions:

Called function must be registered in the current script.

6.5.2 Ret

Syntax:

`Ret``<`***`Ending`***`>`

This instruction returns control to the caller function (or exits the script execution if the function is top-level), but only if the condition specified in *`Ending`* is true. Note that for this instruction, *`Ending`* is *mandatory*.

Example:

`RetNotNeg`

This returns from the current function to the caller in the same script, but only if accumulator is zero or positive.

Length:

1 byte.

Restrictions:

None.

6.5.3 Jump

Syntax:

`Jump``[`***`Ending`***`](<`*`Function_name`*`>)`

This instruction passes the control to the function <*`Function_name`* >. When the jumped-to function returns, control is passed as if current function returned.

Example:

`JumpNotZ``(Fence)`

This gives up control to the function `Fence` in the same script, but only if accumulator is non-zero.

Length:

2 bytes

Restrictions:

Jumped-to function must be registered in the current script.

6.6 Miscellaneous Class Instructions

Instructions in this class are of odd (important, though) variety and are described individually below.

6.6.1 Loop

Syntax:

`Loop`

This instruction begins a loop, which ends with corresponding **`EndLoop`** (see below).

Example:

```
Loop
```

Length:

1 byte

Restrictions:

None.

6.6.2 EndLoop

Syntax:

`EndLoop`

This instruction ends a loop, which begins with corresponding **`Loop`** instruction. It checks if the index register X is greater than 0, and if so, passes control to the instruction immediately following the corresponding `Loop` instruction. It follows from this description that a loop body (i.e., everything between **`Loop`** and **`EndLoop`**) is executed at least once (like FOR in FORTRAN or do/while in C), even if X was negative in the beginning.

Note: **`Loop/EndLoop`** construct uses a fixed loop counter, so nested loops, while technically possible, have limited use and require clever manipulation of the index register. Also, calling a function from within a loop, while legal, is a risky business, because a called function can inadvertently modify the loop counter.

Note: For non-negative X, the loop runs (X)+1 times (provided X is not modified within loop).

Example:

```
LoadX(Literal, 10)
Loop
      MoveI(Literal, 0, TempVar, 3)
EndLoop
```

This clears 10+1 temporary variables 13, 12, 9, …, 5, 4, 3.

Length:

1 byte

Restrictions:

None.

6.6.3 Repeat, RepeatA, RepeatX

Syntax:

`Repeat``(`*`type`*`, `*`num`*`)`
`Repeat``<`*`Reg`*`>`

where `<`*`Reg`*`>` is either **A** or **X**.

These are not real instructions but rather the prefixes modifying the next instruction (of Move Class or Arithmetic and Logic Class only). First, the next instruction is executed the number of times specified in the operand (variable number `num` of type *`type`* for **`Repeat`**, content of Accumulator for **`RepeatA`**, content of index register for **`RepeatX`**); the operand itself remains unchanged (unless changed by the instruction itself). Prefixed instruction executes at least once, even if repeat counter is less than 0. Second, the address of the operand of the next instruction (if it is not of **`Literal`** type) is modified by adding the current repeat count (zero-based).

Using `Repeat` prefix with `Load` instructions seemingly makes no sense. To take advantage of the opportunity to fill the void, the `Repeat` prefix modifies the behavior of `Load` instructions in the following way:
If a `Load` instruction is executed not for the first time, then the previous content of the Accumulator is shifted left by a byte (i.e., `CHAR_BIT` bits), and the least significant byte is filled with the least significant byte of the operand.

Examples:

```
ClearA
Repeat(Literal, 10) Add(InpVar, 1)
```

This calculates the sum of ten input variables 1, 2, …, 10.

```
Repeat(InpVar, 10) Move(Literal, 0x55, OutpVar, 1)
```

This fills a few (namely, whatever number is in input variable 10) output variables, starting with output variable 1, with the hex pattern 55.

```
ClearA
RepeatX XOr(InpVar, 1)
```

This calculates the exclusive OR of input variables 1, 2, …, total of the value of index register.

```
Repeat(Literal, 4) LoadX(Literal, 0x55)
```

Assuming X has 32 bits and CHAR_BIT=8, this loads 0x55555555 to X.

```
Repeat(Literal, 3) LoadA(InpVar, 0)
```

Assuming that A has 32 bits, that **`inpvar_t`** is `signed char` and that input array contains 0xAA, 0xAB, 0xAC, this makes A = 0xFFAAABAC where FF comes from sign-extending the first byte, 0xAA.

Length:

2 bytes (**`Repeat`**), or 1 byte (**`RepeatA`**, **`RepeatX`**).

Restrictions:

Allowed only for Move Class or Arithmetic and Logic Class instructions.

6.6.4 CheckList

Syntax:

`CheckList``(`*`type`*`, `*`num`*`, `*`n`*`, `*`List_type`*`, `*`Arg_List`*`)`
where is a **`CSLIST`**-separated list of numbers:
`Arg_List ::= num | Arg_List` **`CSLIST`** `num.`

This instruction checks whether the value of variable *`num`* of type *`type`* is found among the members of *`Arg_List`* of a specified (namely, *`List_type`*) type. If it finds the match, it puts its 0-based ordinal number in the accumulator; otherwise, it puts a –1. If more than one match is found, the one with the least number is used.
`n` is the number of elements in *`Arg_List`* and is considered a byte-wide literal. *`List_type`* is the common type of elements *`Arg_List`*.
Warning: The number of list argument must be exactly *`n`*. The script **cannot** be executed correctly if it is not so.

Example:

```
CheckList( InpVar, 0, 4, InpVar, 3 CLIST 4 CLIST 12 CLIST 45)
```

This checks whether input variable 0 is equal to one of the four input variables 3, 4, 12, or 45. For instance, if **`InpVar`** 0 is 0x11, and **`InpVar`** 3, 4, 12 and 45 are equal to 0x0f, 0x11, 0x10, 0x11, a value 1 (first match number) is assigned to the Accumulator.

Length:

4 + length-of-the-list bytes

Restrictions:

None.

6.6.5 ComputedCall and ComputedJump

Syntax:

ComputedCall(<*function_name*>)
ComputedJump (<*function_name*>)

These instructions do nothing if accumulator A is negative. If it is non-negative, the **ComputedCall** instruction calls, and the **ComputedJump** instruction jumps to the function, which is registered at the offset A (0-based) from the function <*function_name*>, supplied as the argument. Both instructions can work in concert with **CheckList** instruction: **CheckList** would calculate the offset that **ComputedJump** would use.

Example:

```
RegStart(
      ...
   RegFunc(MyFunc)
   RegFunc(YourFunc)
   RegFunc(HisFunc)
   RegFunc(HerFunc)
      ...
)
...
LoadA(Literal, 2)
ComputedCall(MyFunc)
...
```

In this example, **ComputedCall** will call `HisFunc`, which is at offset 2 from the argument function `MyFunc`.

Length:

2 bytes

Restrictions:

None.

6.6.6 ClearA

Syntax:

`ClearA`

This instruction clears the accumulator register.

Example:

ClearA

Length:

1 byte.

Restrictions:

None.

6.6.7 ClearX

Syntax:

`ClearX`

This instruction clears the index register.

Example:

ClearX

Length:

1 byte.

Restrictions:

None.

6.6.8 Comp1 and Comp2

Syntax:

`Comp1`
`Comp2`

This instruction computes 1's complement or 2's complement respectively of the accumulator register.

Example:

`Comp1`

Length:

1 byte.

Restrictions:

None.

6.6.9 ExchangeAX

Syntax:

`ExchangeAX`

This instruction exchanges value of accumulator and index register.

Example:

`ExchangeAX`

Length:

1 byte.

Restrictions:

None.

6.6.10 MulDiv

Syntax:

`MulDiv`(*`type`*, *`m_num`*, *`d_num`*)

This instruction treats all arguments multiplies the Accumulator by the parameter # *`m_num`* of type *`type`*. The product is then divided by the parameter # *`d_num`* of the same type *`type`*, and the quotient is cast back to **`reg_t`** and assigned to the Accumulator. Any overflow is ignored. Division by zero causes runtime error `CSL_ZERODIVIDE`.

Example:

`MulDiv`(**`Literal`**, `5`, **`Literal`**, `2`)

This multiplies the Accumulator by 2.5.

Length:

3 bytes.

Restrictions:

None.

7. Frequently Asked Questions

7.1 My embedded application does not use C runtime library. Can I still use C-SLang?

Yes. C-SLang doesn't use any runtime library functions exactly because it can target small embedded applications. Nor does it use any standard macros, *except* `CHAR_BIT`, which is `#define`'d in the standard header `limits.h`. If your compiler does not have or does not use this header, you must `#define CHAR_BIT` according to your system's architecture. But `NULL` is `#define`'d in case you do not include standard header(s).

7.2 How many different scripts can run simultaneously?

In principle, you can run any number of C-SLang scripts on any number of C-SLang virtual machines; there is no limitation on C-SLang side.

7.3 My operating environment allows pre-emptive and cooperative tasks. How do I schedule runs of C-SLang scripts?

It's a truism to say that the answer depends on your application. In general, you should use pre-emptive scheduling only if cooperative scheduling is for some reason not sufficient.

C-SLang engine doesn't make use of its own memory, and as such is fully reentrant and safe to use in pre-emptive multitasking.

Whatever the kind of multitasking you choose, make sure that two conditions are met:

- Memory passed to C-SLang to create runtime environment has no conflicts with multitasking, and
- VPIO input and output functions are safe for multitasking

Incidentally, calling **`CSLangExec`** from an interrupt service routine is generally not a very good idea, because interpreting a script can take a while. You may go for it, though, if the script is known to be small and you have a fairly mighty processor. In this case, treat it the same way as pre-emptive scheduling.

Threads should be treated like pre-emptive tasks.

If **`CSLangExec`** takes too much time in cooperative multitasking environment, you may employ debug interface to run the script piecemeal.

7.4 Do I need separate control stacks for different scripts?

No, as long as your scripts cannot run simultaneously. Yes, if the scripts can run at the same time.

7.5 How do I debug a C-SLang script?

Use C-SLang debug interface.

It is a good idea to debug a C-SLang script using a nice integrated development environment, like that of Visual C/C++. For instance, you can execute **`SCLangSetSingleStep`** and **`SCLangSetBreakpoints`** by typing the call statement in the watch window right during debug session. If you plan to use the script in exported format, you can export it right from your debugging session using linked-in **`CSLangBytecodeExport`** function.