BBN Technical Memorandum No. 1221

Sprocket Language Description for the Smart Packets Project

Job No. 11822000

September 27, 1999

Prepared for:
Defense Advanced Research Projects Agency
Computer Systems and Technology Office
3701 North Fairfax Drive
Arlington, VA 22203-1714

Prepared by:
GTE
10 Moulton Street
Cambridge, MA 02138
Sprocket Language Description for the Smart Packets Project

Beverly I. Schwartz

Sprocket is a language created at BBN Corporation for the Smart Packets project. Sprocket was created because no other existing language had a compact enough representation for the Smart Packets environment. Sprocket programs, after compilation and assembly, must fit into single packet that cannot be fragmented, along with packet headers and a data area. Sprocket was designed with the assumption that ethernet will have the limiting MTU. Sprocket is not limited by this assumption; Sprocket will scale up to a larger limiting MTU. Smart Packets programs, by design, will be short; data abstractions such as structures and classes which are essential for managing large software projects, will not be needed in Sprocket.

The philosophy behind the design of Sprocket is that the simplest possible language should be created while not leaving any holes in functionality. Keeping the language simple will allow the encoding for the language to be compact. In addition, Sprocket will have primitives specifically suited for networking environments. These primitives will look like function calls, but unlike a function call, they will map to a single opcode.

Sprocket is compiled into an assembly language called Spanner. Spanner is assembled into a wire encoding. See the document spanner.txt for information on Spanner, and the documents encoding.txt and encoding.gen for information about how Spanner is encoded. (The document encoding.gen is generated when the Smart Packets distribution is built.)

The C-preprocessor is used before the Sprocket compiler parses anything. Sprocket source can be in multiple files, but the files must be included into a primary source file with #include statements. The Spanner code that is produced from one or more Sprocket source files will end up in one Spanner source file. Sprocket source files must have a .spr extension; compilation will produce a Spanner file with the same base name with a .spn extension.

Sprocket is very much like C with a few C++isms. This document will only describe how Sprocket differs from C. Using a reference to the C language and this document, a programmer should be able to put together a Sprocket program.

The rest of this document enumerates the differences between Sprocket and C. Special Sprocket primitives beyond those provided by C will be detailed in the document primitive.txt. Sprocket will not have access to any functions in the standard C library or UNIX system calls.

Sprocket does not use C’s data types. Sprocket’s data types are: unsigned8, unsigned16, unsigned32, unsigned64, unsigned128, signed8, signed16, signed32, signed64, signed128, float32, float64, boolean, string, address, packet, spkt_id, and mib. Variables are declared in Sprocket as they are in C, and they must be declared before they are used.

Variables in a declaration can be initialized with a constant or another variable, but cannot be
initialized with an arbitrary statement. (Initialization from a statement will be added in a future version of Sprocket.) Sprocket recognizes IPv4 and IPv6 address notation for address constants. The boolean constants true and false are also recognized by Sprocket.

Like C++, variable declarations can occur anywhere in the code, intermixed with other statements. Scoping rules and life of variables is the same as C/C++.

Sprocket has complex types which can have actions performed on them. Of the types listed above, string, address, packet, splk_id and mib are all complex types. The actions are Sprocket primitives; the syntax for using these primitives is the same as calling a member function in a C++ class.

Example: The following code declares an instance of type packet and an instance of type address. It then assigns the packet's destination to the address.

```plaintext
packet p;
address a;
a = p.destination();
```

Sprocket does not have any typedef's, struct's, union's, or enum's. Sprocket does not have pointer types. Sprocket allows function parameters to be declared as references (as allowed in C++, using the same syntax). Other than as function parameters, reference variables are not allowed. Sprocket does not support the keywords volatile, register, auto, extern and static. Sprocket does have the concept of a local function variable that holds its value from one call to the function to the next. For this specific use, instead of the word static, the word "persistent" is used.

Sprocket has its own array and list types. To declare an array, a list, or an array of lists, a variable declaration must be preceded with array of, list of, or array of list of, respectively. If the declaration is not provided with a constant initializer, then an array with no dimensions or an empty list results. If the declaration is provided with a constant initializer (uses the same syntax as array initializers in C), then the array or list size is inferred from the constant.

Array elements are accessed the same way they are in C, with square brackets. The size of an array can be set with a Sprocket primitive. Arrays can have a maximum of 3 dimensions. List elements are accessed and manipulated with Sprocket primitives. See the document primitive.txt to learn more about using Sprocket arrays and lists.

Unlike C, Sprocket makes sure an array access is a legal one. If an array access is outside the bounds of the array, then an exception occurs, and the program is exited.

Example:

```plaintext
/* this produces an empty list of unsigned8's */
list of unsigned8 number_list;
/* this produces a 2x3 array of float32's */
array of float32 number_array = { { 1.9, 2.8, 3.7 }, { 4.6, 5.5, 6.4 } };
/* this changes the element that used to have the value 2.8 to value 7.3 */
number_array[0][1] = 7.3;
/* the following statements all would produce exceptions */
number_array[0][5] = 3.3; // 5 is beyond the bounds of the array
number_array[1][-1][2] = 4.8; // negative numbers are illegal for array access
number_array[0][0][2] = 3.3; // too many dimensions for this array
```

Sprocket programs, like C, must have a function main. However Sprocket's main return type is void (not int), and it cannot take any arguments. (Primitives are provided for accessing the Sprocket equivalent of command line arguments.) If main is provided with any arguments, the compiler will not complain, but the assembler will signal an error.

Either C or C++ style comments can be used. (/* */ delimiters or everything from // until the end of the line)
Sprocket supports all C primitives/operators except the ternary operator (?) and the dereference operators (-> and *). Because Sprocket does not support structures, classes or unions, the . operator only has meaning when accessing primitives specific to a Sprocket complex type. The & operator is used only for declaring reference parameters to functions; it is not allowed for dereferencing variables.

Sprocket does automatic type promotion when different numeric types are intermixed within arithmetic, relational and bitwise operations. The "lesser" type is promoted to the "greater" type. The order of promotion, from least to greatest is: boolean, unsigned8, signed8, unsigned16, signed16, unsigned32, signed32, unsigned64, signed64, unsigned128, signed128, float32, float64, string.

Sprocket does automatic casting with equivalence operators. The type to the right of an equivalence operator is cast to the type to the left of the equivalence operator. The compiler will issue a warning if a greater type is cast to a lesser type in an equivalence statement.

Sprocket allows explicit casting, as in C. If a particular cast is not allowed, an exception is generated and the program is terminated. (e.g. A numerical type cannot be cast to a packet.)

Addresses can be cast to numerical types. If addresses are used in operations which do automatic type promotion, they are converted to an unsigned type of the correct size (unsigned32 for IPv4 addresses, unsigned128 for IPv6 addresses) before automatic promotion is done. Same goes for both implicit and explicit casts; the address is converted to an unsigned integer before the cast occurs.

Example:

    unsigned8 result;
    string s = "el";
    unsigned16 u = 2;
    float32 f = 3.244;
    result = (float32)(f + u + s); // result ends up with the value 52
    // u is converted to the floating point number 2.0
    // f + u produces the value 5.244
    // 5.244 is converted to the string "5.244"
    // "5.244" + s produces the string "5.244\el"
    // because of the explicit cast to float32, the string "5.244\el" is
    // converted to the value 52.44
    // the value 52.44 is cast to an unsigned8, producing the value 52
    // the value 52 is put into result

Sprocket supports all C control flow keywords except for goto. Sprocket requires curly brackets with all control flow statement blocks, including single statements.

Sprocket allows loops to be labeled; break and continue statements can be followed with a label to indicate which loop the break or continue applies to. The syntax for labels is to put a colon and the label immediately before the opening curly bracket for the statement block. Loop labels are not required for loops or for the break and continue statements. If break or continue are not followed by a label, then the break or continue applies to the innermost loop.

Example:
/* index counters */
unsigned16 i, j, k;
/* make a 3-dimensional (3x4x2) array */
array of signed16 dim3_array = {
    { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } },
    { { 9, 10 }, { 11, 12 }, { 13, 14 }, { 15, 16 } },
    { { 17, 18 }, { 19, 20 }, { 21, 22 }, { 23, 24 } }
};
/* look for an array element of value 14; exit loops when found */
for (i = 0; i < dim3_array.dimension_size(0); i++) : outermost_loop {
    for (j = 0; j < dim3_array.dimension_size(1); j++) {
        for (k = 0; k < dim3_array.dimension_size(2); k++) {
            if (dim3_array[i][j][k] == 14) {
                break outermost_loop;
            }
        }
    }
}

Sprocket requires function prototypes before functions are used, as in ANSI C.

The rest of this document is an example of a full Sprocket program. For each argument supplied as command arguments, it calculates its fibonacci number. It puts the results of all of the calculations in a packet and sends the packet back to the originating host.
function prototypes for calculating fibonacci numbers
algorithm comes from "Structure and Interpretation of Computer Programs"
by Harold Abelson and Jay Sussman, page 39

unsigned64 fib(unsigned8);
unsigned64 fib_iter(unsigned64, unsigned64, unsigned64);

void main()
{
   /* declare an instance of packet to be sent back to source */
   packet pkt;

   /* get the number of arguments supplied to the program */
   unsigned8 num_args;
   num_args = spkt_num_args(); // a Sprocket primitive

   /* put how many arguments we have into the packet */
   pkt.data_append(num_args); // another Sprocket primitive

   /* calculate the fibonacci number for each argument */
   unsigned8 count;
   for (count = 0; count < num_args; count++)
   {
      /* get the argument (all arguments are strings) */
      string s;
      s = spkt_arg(count); // another Sprocket primitive

      /* convert the argument to an unsigned8 */
      unsigned8 n;
      n = s;

      /* calculate the fibonacci number using our fib function */
      unsigned64 fib_result;
      fib_result = fib(n);

      /* put the number and the result into the packet */
      pkt.data_append(n); // we've seen this Sprocket primitive
      pkt.data_append(fib_result);
   }

   /* send the results back to the originating host */
   pkt.send(); // yet another Sprocket primitive
}

/* for calculating fibonacci numbers */
unsigned64 fib(unsigned8 n)
{
   unsigned64 a = 1;
   unsigned64 b = 0;

   /* use this different function to prevent huge amounts of recursion */
   return fib_iter(a, b, n);
}

/* the anti-tail-recursion fibonacci function */
unsigned64 fib_iter(unsigned64 a, unsigned64 b, unsigned8 count)
{
   if (count == 0) {
      return b;
   } else {
      return fib_iter(a + b, a, --count);
   }
}