Krynen Technology

The SF programming language

Introduction

SF is an object oriented generic programming or hardware definition language for embedded purposes. The name SF was originally meant as an acronym for Structured Forth, but the SF syntax has no resemblance with the Forth programming language syntax and the name is just SF now without hinting to Forth. The SF compiler will be supported by an IDE named ***Prime*** which includes a code/design browser, simulator and (target) debugger. The 'structure' in the SF former name Structured Forth did not imply that Forth has no structure or does not offer structure but that the SF source code layout is strict (*like coding standard rules*) to improve readability and more modularity and clarity is offered by data structures, strong typing and object oriented techniques.

Forth is a member of the class of extensible - no distinction between core words and created routines - interactive languages. A Forth compiler-interpreter can be seen as an extensible full macro-assembler and integrated operating system for an abstract stack processor (*virtual machine*). Forth's postfix notation and its inner construction allows very compact refactored code providing a terminal, an editor, a compiler and an interpreter within 16KB of code for a general 8-bit processor like the Z80 or 6502. The low memory requirement and the interactive debugging feature were a great advantage in the beginning years of programming microprocessors.

SF is intended as a very readable object oriented generic programming or hardware definition language while keeping the unique features from Forth as much as possible. The inner works of SF are still very Forth-like. Infix statements and function calls are translated to postfix statements before being processed. The SF cross-compiler generates as a programming language depending on target system requirements - speed vs code size - native, indirect or token threaded code for a (simulated) micro or soft core processor. As a hardware definition language (HDL) the compiler generates EDIF for programmable logic e.g. an FPGA. A subset of the SF language is supported for HDL synthesis.

Contents

- Functions and their declarations
- Functions objects and delegates
- Casting
- Number notation and radix
- Blocking assignments
- Conditional blocking assignments
- Non-blocking assignments
- Operators
- Statements
- Modules
- Classes
- Enums and sets
- Structs and unions
- Dependencies
- Stack manipulations
- Contract
- Control structures
- Algebraic assembly
- Exception handling
- Templates
- Concurrency
- Memory management
- Garbage collection

Build process

The SF cross-compiler is an incremental compiler evaluating all source code (*global analysis*) used as input for complete - all - target code generation. There are no forward reference declarations required. The SF compiler keeps track of all references (*in-memory indexes*) by means of a low level database (*no query language*) also written to disk as a repository for rebuilds, dynamic loading and debugging. Some file extensions are reserved for SF processing:

.sf	-	source code
.sfm	-	the **main** source code
.sfp	-	processed source code, offers interface (public), dependencies (use) and object code (hexadecimal text format)
.sfl	-	library of processed source code, contains multiple *.sfp files
.sfr	-	repository of the actual build process, also used as input for rebuilds, dynamic loading and debugging

Interpreter

The interpreter is invoked when the `'#'`-token is found on the first position of a line. This is the SF interpreter on the cross-compiling **host** system, not on target! Invoking the host system interpreter allows tailoring of the compilation process.

The interpreter on target could be used for debugging and is accessed via a target shell command line tool. The target interpreter does not compile but it does provide e.g. execution and dynamic loading of SF modules, classes and functions.

Indentation

The indentation is used to group statements and is part of the SF programming language. Before the first token tabs are allowed or otherwise the number of matching spaces, but no combination of tabs and spaces. After the first token there are no tabs allowed, only spaces.

Line joining does not affect the logical indentation structure. Within a logical line the open and closing parenthesis should match even if its spans multiple physical lines. Thus, parenthesis could be used to extend a statement or an expression beyond a single physical line. String literals also could be spanned over multiple lines. The indentation of the 'spanned' lines is incremented compared to the first physical line and in case of string literals the indentation equals a single space (is added between line separated text).

Curly braces - `'{'`, `'}'` - and empty lines are optional. They could be added to improve readability. The curly open and close braces should match - e.g. `{{}{}}` - and have the same statement indentation (as statements preceding the curly open brace). Between statements only a single empty line is allowed. Lines with curly braces are regarded as empty statements, however they could also be used as defining the scope of compound statements and life cycle of local variables:

```
.
{
  mutex : Mutex // Local Mutex constructor.
  mutex.take
  .
  .
  mutex.release
} // Mutex destructor.
.
```

Token delimiters

1st stage : `' ' '.' '\n'`

2nd stage : if 1st stage delimiter = `' '` or `'\n'` then check for:
- token last character = `',' '(' '[' '*' '<' '>'`
- token first character = `'*' '"' '\'' '<' '>'`
- special sequences like: `((`, `(((`, `)))`, `))`, `..`, etc.

3rd stage : if 2nd stage token last character = `'*'` then check for:
- token second last character = `'*'`

if 2nd stage token first character = `'*'` then check for:
- token second character = `'*'`

if all tokens are checked, check for:
- module, class or function
- number, see number notation
- templates

The last successful check for number or function sets the search order for the 2nd stage of the next token. If the last token was a number then the 2nd stage will check for number first and visa versa if the last token was a function. The `'.'` delimiter is a scope delimiter for structs, classes and modules.

Comment definition

Single line : `//`
Multiple lines : `/* .. ..... ..... .. */`

Name notation

Input text should be UTF-8 compliant, multibyte unicode characters are only allowed within comments or strings. Names of struct, enum, alias and type definitions are written in capitals (uppercase letters), except for an optional underscore to separate capital words or an optional underscore separated prefix for lowercase (e.g. hungarian) notation characters. Names of classes and modules start with a capital and are build from letters, no underscores allowed. Abstract classes start with a capital `'I'`. Variable and function names start with a lower case character and are build from letters. For class method names almost all screen visible 7-bit ASCII characters are allowed. If the class method starts with a letter it should be a lowercase letter. The class method name should not contain token delimiters. Identifiers have a maximum length of 255. Case is significant. However, case-insensitive duplicates are not allowed.

Type definition

Format single line type definition:

```
alias <new type> = <existing type>
```

Using alias: type checking is based on <existing type>, it allows weak typing.

```
type <new type> = <existing type>
```

Using type: type checking is based on <new type>, a type cast has to be used when a check is done against the <existing type>. This supports strong typing.

Format multi line type definition

```
type
  <new type> = <existing type>
  <new type> = <existing type>
  <new type> = <existing type>
  .
  .
```

Examples:

```
type UNSIGNED_8BITS = BYTE
type UNSIGNED_8BITS = U8
type SIGNED_8BITS   = S8

type
  B1                = B[ 1 ]
  B8                = B[ 7 : 0 ]
  U12               = U[ 11 : 0 ]
  U16               = U[ 16 ]
  U_FIX16_FRAC16    = U[ 0.16 ]
  S32               = S[ 31 : 0 ]
  S_FIX32_FRAC29    = S[ 3.29 ]
```

```
   UNSIGNED_16BITS   = U16
   SIGNED_16BITS     = S16
   UNSIGNED_32BITS   = U32
   SIGNED_32BITS     = S32

 type SIGNED_64BITS  = S64
 type REAL_32BITS    = R32 // Hardware floating point.

 type
   REAL_48BITS       = R48 // Software floating point emulation.
   REAL_64BITS       = R64 // Hardware floating point.

 type BIT_BOOLEAN    = B1  // Single bit boolean.
 type BYTE_BOOLEAN   = B8
 type WORD_BOOLEAN   = B16
 type DWORD_BOOLEAN  = B32
 type SINGLE_CHAR    = CHAR
 type UNSIGNED_8BITS = CHAR

 type
   255CHAR_STRING     = STR255          // Dynamic string.
   20CHAR_STRING      = STR255( 20 )    // Static string of maximum 20
characters.
   64K_MINUS_2_STRING = STR64K          // Dynamic string.
   1000CHAR_STRING    = STR64K( 1000 )  // Static string of maximum 1000
characters.
   4G_MINUS_4_STRING  = STR4G           // Dynamic string.

 # if uP8051                           // Interpreter on host invoked.
 #   type STRING = STR255
 # if uP8086
 #   type STRING = STR64K
 # if uP80386
 #   type STRING = STR4G
```

For the SF built-in types **S** (signed) and **U** (unsigned) memory or logic register a fraction could be defined (Q format - number after dot represents fraction bit width, sum of numbers represents register bit with). This affects calculations, but it could also clearify its meaning e.g. U[0.16] full scale value = 0 dB, half scale value = -6 dB. The LSB value for the register type is optional, it is always zero.

Array type definition

```
 type SINGLE_DIMENSION  = U8[ 10 ]
 type TWO_DIMENSION     = U8[ 10 ][ 20 ]
 type
   MULTI_DIMENSIONS_U8  = TWO_DIMENSION[ 1000 ]
   STRING_ARRAY10       = STRING[ 10 ]  // Static array of 10 strings of type
                                        // STRING.
   DYNAMIC_S16_ARRAY    = S16[ ]        // Dynamic array.
   HASH_TABLE           = DYNAMIC_S16_ARRAY[ 10 ]
                                        // Fixed hash table, dynamic array
                                        // length.
   DYNAMIC_MULTI_ARRAY  = U32[ ][ ]     // Dynamic two-dimensional array.
```

Dynamic arrays are structs containing a cardinal reflecting the total allocated memory on the heap and a reference to this allocation.

Pointer type definition

```
type U32_POINTER = *U32     // *U32 reads from left to right as pointer-to-
                            // type-U32.
type
  ARRAY_POINTER  = *TWO_DIMENSIONS
  STRING_POINTER = *STR255( 100 )
  // Warning *CHAR is not a string type, it is a pointer to a single char.
  // Dynamic arrays and dynamic strings are pointers.
  S64_PTR_PTR    = **S64
```

Other type definitions

```
struct <new struct>
enum   <new enum>
class  <new class>

struct BITBOOLEANS          // Memory reservation depends on uP architecture,
{                           // type indication is optional [].
  timer0Active : B1
  timer1Active : B1
  timer2Active : B1
}

struct BITBOOLEANS : U16    // Reserves 16 bits for this struct.
{
  ...
}

struct BITBOOLEANS : U32[ ] // Reserves multiple of 32 bits for this struct.
{
  ...
}
```

Variables, constants, parameters and pointers

```
size : U16 = 0
const FALSE = 0
const NR_CHANNELS = 3
const CH_SELECT : B[ 2 : 0 ] = 4
pData : *U8 = NULL
object SpiSelect(
# param NR_MODULES = 5     // Default value
# param SPI_DATA_WIDTH = 8 // Default value
  IN  clk : B1,
  IN  spi_m : B[ SPI_DATA_WIDTH ],
  IN  spi_m_valid : B1,
  OUT spi_s : B[ SPI_DATA_WIDTH],
  OUT spi_s_valid : B1,
  IN  spi_active : B1,
  IN  spi_status : B[ SPI_DATA_WIDTH ],
  OUT spi_rd : B1,
  OUT spi_wr : B1,
  OUT spi_addr B[ 14 ] spi_addr,
  IN  spi_data_rd : B[ SPI_DATA_WIDTH ] // Should be 0 when select = 0
  OUT spi_data_wr : B[ SPI_DATA_WIDTH ],
  OUT select : B[ NR_MODULES ],
  ) : // Output
  spi_int : B1
```

Dereferencing variables and pointers

Pointers and pointer-pointers are automatically dereferenced by the compiler. Examples:

```
 struct s_MSG
 {
   size  : U16
   pData : *U8
 }

 sendMessage( IN message : *s_MSG ) : BOOL
 {
   pre ( message* <> NULL ) // Dereferencing on stack (pointer).

   if ( message.size = 0 )  // Pointers are automatically dereferenced by
                            // compiler.
   {
     return FALSE
   }
   .         // "*s_MSG" reads from left to right as pointer-to-type-s_MSG.
   .         // "message*" reads from left to right as message-pointer.
   .         // "*message" gives the address of the stack parameter,
   .         // reads from left to right as pointer-to-message!
 }

 sendMessage( IN message : **s_MSG ) : BOOL
   pre
     message** <> NULL  // First dereferencing on stack (pointer-pointer).
     message*  <> NULL  // Second dereferencing on pointer.

   if message.size = 0  // Pointer-pointers are automatically dereferenced by
compiler.
     return FALSE
   .
   .         // "**s_MSG" reads from left to right as pointer-pointer-to-type-
             // s_MSG.
   .         // "message**" reads from left to right as message-pointer-pointer.

 {
   msg1 : s_MSG
   msg2 : *s_MSG  = *msg1
   msg3 : **s_MSG = *msg2

   // Automatic dereferencing by compiler.
   write( "%b", msg1.size = msg2.size )  // TRUE.
   write( "%b", msg1.size = msg3.size )  // TRUE.

   u8var  : U8  = 0
   pU8var : *U8 = *u8var  // *u8var reads from left to right as pointer-to-
                          // u8var.
   invariant
     pU8var* <> NULL      // pU8var* reads from left to right as pU8var-pointer.

   if pU8var > 5          // Automatic dereferencing by compiler!
     return FALSE

   var1  : U8   = 1
   var2  : U8   = 2
   pVar1 : *U8  = *var1               // Assignment address var1.
   pVar2 : *U8  = *var2               // Assignment address var2.
   ppVar : **U8 = *pVar1              // Assignment address pVar1.
   var3  : U32  = U32( pVar1* )       // Assignment cast content pVar1 = *var1.
   var4  : U32  = U32( *pVar1 )       // Assignment address cast pVar1.
```

```
   write( "%b", pVar1* = *var1 )       // TRUE, pVar1-pointer equals address
                                      // var1.
   write( "%b", ppVar* = *pVar1 )     // TRUE, ppVar-pointer equals address
                                      // pVar1.
   write( "%b", ppVar** = *var1 )     // TRUE, ppVar-pointer-pointer equals
                                      // address var1.
   write( "%b", *U8( var3 ) = var1 )  // TRUE, var3 contains pointer and cast
                                      // dereferencing equals var1.
   write( "%b", **U8( var4 ) = var1 ) // TRUE, var4 contains pointer-pointer
                                      // and cast dereferencing equals var1.

   write( "%b", pVar1 = 1 )           // TRUE, pVar equals 1.
   POINTER( *pVar1 ) := *var2         // Pointer assignment!
   write( "%b", pVar1 = 2 )           // TRUE, pVar equals 2 now.
   pVar1 := var1                      // Compiler assigns var1 to var2.
                                      // <*U8> := <U8> is accepted, because
                                      // compiler automatically dereferences
                                      // pVar to var2.
   write( "%b", pVar1 = 1 )           // TRUE, pVar equals 1 again.
   write( "%b", var1 = var2 )         // TRUE, var1 equals var2.
   pVar2 := 2
   write( "%b", var2 = 2 )            // TRUE, var2 equals 2 again.

   write( "%b", ppVar = 1 )           // TRUE, ppVar equals 1.
   POINTER( *ppVar ) := *pVar2        // Pointer assignment!
   write( "%b", ppVar = 2 )           // TRUE, ppVar equals 2 now.
   ppVar := var1                      // Compiler assigns var1 to var2.
                                      // <**U8> := <U8> is accepted, because
                                      // compiler automatically dereferences
                                      // ppVar to var2.
   write( "%b", ppVar = 1 )           // TRUE, pVar equals 1 again.
   write( "%b", var1 = var2 )         // TRUE, var1 equals var2.
   ppVar := 2
   write( "%b", var2 = 2 )            // TRUE, var2 equals 2 again.

   ppVar := *var2                     // Error, type mismatch (U8 <> *U8). The
                                      // pp-prefix
                                      // helps to remind that the variable has
                                      // a pointer-pointer type but the
                                      // compiler dereferences!.
}
```

As a general rule variables that are pointers to types are automatically dereferenced. Pointers to variables, contents of pointer or pointer-pointer variables are not automatically dereferenced. E.g. `ppVar` is a variable which is a pointer-pointer-to-type-U8 and is automatically dereferenced while `ppVar**`, the actual content of the variable is not! Outside type declarations pointer `*` or pointer-pointer `**` type cast are also automatically dereferenced! The `POINTER` type exist to allow pointer assignments and pointer arithmetics.

Strings

Three types of strings are defined: `STR255`, `STR64K` and `STR4G` with respectively maximum 255, 64K - 2, and 4G - 4 characters. Those string types could be defined dynamic as well as static, e.g. `STR255` or `STR255( 10 )`. Static defines reserves the required memory space. Dynamic strings are automatically converted to static strings if their size and content does not change during their life cycle. Special characters could be inserted with the escape `'\'`-character e.g. `'\n'` for line feed. Literal strings are possible as well by means of a `'*'`-prefix:

```
winSysPath : STR255 = *"c:\windows\system" instead of
winSysPath : STR255 = "c:\\windows\\system"
```

For string literals `'"'`-characters are inserted with double `'"'`-characters : `""`. Literal strings could be considered as processed strings located somewhere in memory, hence the prefix `'*'`, pointer-to-string. String catenation is possible with the `'+'`-operator:

```
winSysPath := "c:\\windows" + "\\system" or
winSysPath := "c:\\windows"
winSysPath.+( "\\system" )              // Class method style.
```

On construction strings are always initialized empty if there is no initialization done.

Functions and their declarations

Parameters and result are put on stack from left to right. Parameter `'IN'`, `'INOUT'`, `'OUT'` parameter prefix allow the compiler to check what kind of actions are allowed on the parameters. Parameters by reference are already allocated objects or memory. Objects or memory allocated inside the function body are not allowed to pass (as reference) via `'INOUT'` or `'OUT'` parameters, but should be passed as a result of the function - dynamic allocations as reference and stack allocations as value (copy). `'INOUT'` parameters are always references (*-prefix types or pointer-types). `'OUT'` parameters are references only when they are allocated before the call. The exception for these rules are the function **object**s, because those objects keep their internal state.

```
// s_MSG struct is allocated before call.
sendMessage( IN message : *s_MSG ) : BOOL  // IN is default.

// S32 variable is allocated before call.
incrementInteger( INOUT int : *S32 )

// s_MSG struct is allocated before call.
receiveMessage( OUT message : *s_MSG ) : BOOL

// s_MSG struct is allocated within call.
receiveMessage( IN handle : *MSGQUEUE ) : e_ERRORMSG, *s_MSG, BOOL
{
  msg        : *s_MSG     = new( s_MSG )   // s_MSG constructor.
  error      : e_ERRORMSG = OK
  successful : BOOL       = FALSE
  .
  .
  successful := TRUE
  .
  if handleTimeout
  {
    error      := e_HANDLE_TIMEOUT
    successful := FALSE
  }
  .
  // The order of result is significant. It should match the function
     declaration.
  return error, msg, successful
}

receiveMessage( ...
{
  .
  .
```

```
  // The order of result is significant, however multiple pushes are
  // possible . The actual return happens when it gets out-of-scope.
  push error, msg
  .
  .
  push successful
  .
} // Return gets out-of-scope here.
```

Example:

```
message : *s_MSG
error   : e_ERRORMSG = OK

if message, error := receiveMessage()
{
  // Process receiving message.
}
else
{
  case error
  {
    case e_HANDLE_TIMEOUT
      // Handle timeout.
      break
      .
      .
    default
  }
}

public overload getNumberOfEntries( database : *DB ) : U32
public overload getNumberOfEntries( database : *DB,
                                    indexKey : STR255 ) : U32

public delay10ms

public overload logMsg( logMsg : STR255 ) : BOOL
// logMsg( "Test stdout" ) default output stdout

public overload logMsg( device : devId,
                        logMsg : STR255 ) : BOOL
// logMsg( socket, "Test socket" )

#define WAIT_FOR_ACKNOWLEDGE TRUE  // Wait for return function.
public overload logMsg( device : devId,
                        logMsg : STR255,
                        param  : U32[ ], // dynamic array!
                        ack    : BOOL = FALSE ) : BOOL

// Default parameters implies also overloading:
// logMsg( socket, "Test socket %d, %d, %d", ( 1, 2, 3 ))
// logMsg( socket, "Test socket %d, %d, %d", ( 1, 2, 3 ),
           WAIT_FOR_ACKNOWLEDGE )

previous( IN number : U32 ) : U32
{
  decrement( number ) // Number on stack is modified.
  return number
}

// Alias works also for functions, it works for every reference.
```

```
alias prev = previous
```

Function objects and delegates

If a function is declared as object, the function parameters and internal variables are stored as an object. On return the actual values of parameters and internal variables keep their actual values. On repetitive calls the function continues where it has returned previously until the function exits or is destroyed. Because the function represents an object the first letter of the function starts with a capital - like a class or module - and is only to call through a delegate.

```
public object ReadCharFromBuffer( IN buffer : CHAR[ ] ) : CHAR
  getChar  : *CHAR   = buffer    // Assignment of array with known size to
                                 // pointer. Triggers
                                 // SF compiler to check array bounds on
                                 // getChar*.
  pGetChar : POINTER = *getChar
  try
    while getChar <> 0           // getChar is dereferenced automatically!
      return getChar             // Delegate returns, but continues on next
                                 // line!
      inc( pGetChar )            // pGetChar is dereferenced automatically!
                                 // Or "inc( POINTER( *getChar ))". No pGetChar
                                 // required.
                                 // The pointer arithmetic is allowed because
                                 // pGetChar has been
    return 0                     // assigned a pointer or POINTER() represents
                                 // a pointer where
  catch                          // the bounds (usually from an array) are
                                 // checked.
    return                       // Ignore exception, getChar* points outside
                                 // the array.

buffer         : CHAR[ 100 ] = ( 1..99, 0 )
getChar        : ReadCharFromBuffer( buffer )  // Declare delegate.
charFromBuffer : CHAR

do
  charFromBuffer := getChar()
  .
  .
while charFromBuffer <> 0

// Or catch exception when the getChar() delegate becomes invalid (= NULL).

try
  loop
    charFromBuffer := getChar()
    .
    .
catch
  return // Ignore exception.
```

Casting

As general rule casting is implicit if no information is lost, otherwise it is explicit. Pointer casting is always explicit.

```
counter1 : U16 = 0
counter2 : U8  = U8( counter1 ) // Downcast is explicit required.
```

Cast functions could be defined:

```
public cast U8( unsigned16bit : U16 ) : U8
  return unsigned16bit[ 7 : 0 ] // ':' indicates bit range.
```

The **cast** specifier triggers a compiler check that the function name is a class or type name and the result equals this type name. Pointer casting requires no specific cast functions.

```
u8array : U8[ 10 ]  // Assume last 4 bytes reserved for a crc32 value.
crc32   : U32  = calculateCRC32( *u8array, u8array[ ( sizeof( u8array ) -
                 sizeof( crc32 )) ] )
pCrc32  : *U32 = *U32( *u8array[ ( sizeof( u8array ) - sizeof( crc32 )) ] )

if crc32 = pCrc32
  return TRUE
```

Number notation and radix

Default decimal. Prefix zeros are ignored. First character starts with `'0'..'9'`, `'-'`, `'.'` or #.

```
'b110110100     // Binary.
10'b1           // Binary 10-bit.
'o73110         // Octal.
'h0EF3067AB     // Hexadecimal.
0xEF3067AB      // Hexadecimal.
0123456         // Decimal.
-.8999          // Floating point.
0.15            // Floating point.
25.             // Floating point.
3e10            // Floating point.
R48( -3.5E-23 ) // Floating point, type R48.

EF3067AB        // Error, no distinction from class possible.
```

Blocking assignments

Blocking assignments are assignments executed immediately. They are not delayed.

Single line, multi (chained) assignments are possible. The token `':='` is in fact a class method according to the format `<class>.<method>( IN input : <class> )`. This assignment format together with the operator format allows infix notation.

```
counter1 := counter2 is equivalent to counter1.:=( counter2 )
```

Infix notation requires two operands and an operator. Assignment could be considered as a special kind of operation: `<l-value> := <r-value>`. The compiler treats the `l-value` as an object by reference and the `r-value` as an input object by value. During parsing both the `l-value` as `r-value` are placed as references on the compiler' internal stack. The reference of the `r-value` is used to retrieve the content and place it on the data stack, while the reference of the `l-value` is used for assignment.

```
// Counter initialization.
counter1, counter2, counter3 := 0
```

Three object references are put on the compiler' internal reference stack and the `':='`-method is applied to all of them with `0` as input.

```
// receiveMessage( IN handle : *MSGQUEUE ) : e_ERRORMSG, *s_MSG, BOOL
```

```
error      : e_ERRORMSG
msg        : *s_MSG
successful : BOOL

error, msg, successful := receiveMessage( msgHandle )
```

or within a conditional statement:

```
if ( error, msg := receiveMessage( msgHandle )) // 'if' processes BOOL on
                                                // stack.
{
  .     // If TRUE error and msg are assigned. This means both
  .     // parameters are popped from stack.
}
else
{
  pop   // Pop '*s_MSG' from stack (and lost).
        // Now error parameter could be popped from stack.
  writeln( LTP1:, "Receive message error, error = %d", pop )
}
```

The token `':='` is reserved for format `<class>.<method>( IN input : <class> )`. Every class method declaration of `':='` should match this signature.

Conditional blocking assignments

Conditional blocking assignments are assignments executed immediately based on a condition. They are not delayed.

```
// receiveMessage( IN handle : *MSGQUEUE ) : e_ERRORMSG, *s_MSG, BOOL
error, msg ?= receiveMessage( msgHandle )
        // '?=' processes BOOL on stack.
  .     // If TRUE error and msg are assigned. Both
  .     // parameters are always popped from stack.
```

The token `'?='` is reserved for format `<class>.<method>( IN input : <class> )`.

Non-blocking assignments

Non-blocking assignments are selective assignments delayed until end of function or process.

Single line, multi (chained) assignments are possible. The token `'<='` is in fact a `':='` class method according to the format `<class>.<method>( IN input : <class> )`. This assignment format together with the operator format allows infix notation.

```
counter1 <= counter2 is equivalent to counter1.<=( counter2 )
```

Infix notation requires two operands and an operator. Assignment could be considered as a special kind of operation: `<l-value> <= <r-value>`. The compiler treats the `l-value` as an object by reference and the `r-value` as an input object by value. During parsing both the `l-value` as `r-value` are placed as references on the compiler' internal stack. The reference of the `r-value` is used to retrieve the content and place it on the data stack, while the reference of the `l-value` is used for assignment.

```
// Counter initialization.
counter1, counter2, counter3 <= 1
```

Three object references are put on the compiler' internal reference stack and the `'<='`-method is applied to all of them with `1` as input.

When multiple non-blocking assignments to the same object are stated within a function or process, only the last non-blocking assignment will be executed:

```
 // Assume counter1 has an actual value of 1.
 counter1 <= counter1 + 2
 if counter1 = 1
 {
     counter1 <= counter1 + 1 // counter1 gets the value of 2 and not 3,
statement
     .                        // "counter1 <= counter1 + 2" is not executed!
 }
```

Blocking and non-blocking assignments to the same object are not allowed. The token `'<='` is reserved for assignments as described with format `<class>.<method>( IN input : <class> )`.

Operators

Class methods with the format `<class>.<method>( IN input : <class> ) : <class>` and `<class>.<method>( IN input : <class> ) : BOOL` are recognized as operators by the compiler. Recognized as operators, class methods could be expressed in infix notation. The compiler interprets the `l-value` as an object by reference of the first encountered (most left) class method and the `r-value` as an input object by value:

```
class U32
{
  varU32 : BYTE[ 4 ]
   .
   .
  public +( input : U32 ) : U32  // Add operator.
    asm
      eax := varU32              // mov EAX, varU32
      eax.+( input )             // add input
      push eax
}

  var1.+( var2 )         // Class method call.
```

This could make method call constructions possible like:

```
  var1.+( var2.+( var3.+( var4 )))
```

The variables `var2` and `var3` are modified as well! The infix statement

```
  var1 := var2 + var3 + var4
```

will be processed by the compiler as postfix operations: `(( var4 ) var3 + ) var2 + ) var1 := ---> (value var4) (reference var3, value copied on stack) <+> <results value on stack> (reference var2, value copied on stack) <+> <results value on stack> (reference var1) <:=>.`

Using the infix notation the compiler understands that the variables `var2`, `var3` and `var4` are not to be modified, but their references are used to make temporary copies on stack and these copies are

modified! But, this does not apply for the first encountered class method `':='` where the reference is used to modify the object itself.

So, except for the first encountered method the infix notation instructs the compiler to copy the content (object) of the `l-value` reference on the data stack and to process the method on the stack reference. In case of methods or functions the result should match the class. However, if the method or function is placed as first token the result should match a valid reference to a class object:

```
getElementReferenceFromArray( 2 ) := var1 + var2 + var3
```

There is no operator (class method) precedence. The compiler translates infix statements internally to postfix statements and processes them accordingly. The order of processing could be affected by using parenthesis:

`var1 := var2 * var3 / 4` is equivalent to `var1 := ( var2 * var3 ) / 4 )` while `var1 := var2 * (var3 / 4 )` might be wanted as result.

If no parenthesis are applied and one or more operator(s) are found within a the `r-value` of a single statement, the compiler generates a warning `"W18: Ambiguous interpretation possible! Apply parenthesis!"`. This warning generation could be ignored by an interpreter instruction in the operator definition:

```
class U32
{
  varU32 : BYTE[ 4 ]
    .
    .
  public +( input : U32 ) : U32  // Add operator.
# this.ignore( W18 )             // Ignore warning W18 during compilation.
    asm
      eax := varU32              // mov EAX, varU32
      eax.+( input )             // add input
      push eax
}
```

To prevent generation of this warning all operators found within the single statement should ignore this warning. If a method has an assignment or operator signature and infix notation is not allowed the compiler should be directed to disallow this:

```
class NoInfix
{
    .
  public +( input : NoInfix ) : NoInfix
# this.noInfix                   // Infix notation not allowed.
    .
}
```

The format `<class>.<method>( IN input : <class> ) : <class>` is reserved for binary arithmetic operators like `'+'`, `'-'`, `'/'`, `'*'`, etc. and bitwise operators like `'and'`, `'or'` etc. as the format `<class>.<method>( IN input : <class> ) : BOOL` is reserved for boolean operators like `'>'`, `'='`, `'>='`, etc., but also specifically the format `BOOL.<method>( IN input : BOOL ) : BOOL` for logical `'and'`, `'or'`, `'xor'`, `'not'`, etc. Every class method definition of those operators should match these signatures.

Due to how operators are defined (allowance infix notation) and the left-to-right parsing where the most left token (`l-value`) is always regarded as a reference there are three equivalent ways to describe an operation on the most left token:

```
 (1) var1 := var1 + 5
 (2) var1.+( 5 ) and
 (3) var1 + 5

 (1) var1 := var1 + var2 + 5
 (2) var1.+( var2 + 5 ) and
 (3) var1 + ( var2 + 5 )          // Parenthesis are optional, but in this case
recommended.
```

Statements

Per logical line one - single - statement is allowed. Every statement should have a balanced data stack, except for stack operations like **return**, **push** and **pop**. Compound statements are grouped statements with the same indentation.

Modules

A **module** is a container for classes and functions. The name of the module should match the file name without extension. The module defines the scope of classes and functions it contains: `<module>.<class>` or `<module>.<function>`. A module could hold more than one class, but if there is a single class then defining a module is optional. In case of a single class and no module definition the class name should match the file name without extension. All class methods, functions, enums and variables are private by default - there scope is within the module or class. Variables are never allowed to become public. Only functions or class methods could be exported as public interface by the reserved word **public**. Variables are accessible through functions or class methods who could have the same name as the variable.

Example:

```
module Console

use Mutex

public struct s_WINDOW
  screenWidth  : U8 = 80
  screenHeight : U8 = 25

mutex  : Mutex
window : s_WINDOW

public overload window : s_WINDOW   // Read access.
  mutex.take
  push window                       // Push window on return stack.
  mutex.release

public overload window( inWindow : s_WINDOW )     // Write access.
  mutex.take
  window := inWindow
  mutex.release

public overload window( width : U8, height : U8 ) // Write access.
  mutex.take
  screenWidth  := width
```

```
   screenHeight := height
   mutex.release
```

If a variable is overloaded the compiler checks in case of read access that no parameters are defined and indeed the variable itself is returned and in case of write access indeed only the variable itself is modified via `'IN'` parameters. `'INOUT'` or `'OUT'` parameters are not allowed. If the write access is omitted there is just read-only access. Fields or member data have automatically variable scope.

Classes

A class, or class type, defines a structure consisting of fields, methods, and interface. Instances of a class type are called objects. A field is essentially a variable that is part of an object. Like the fields of a struct, class fields represent data items that exist in each instance of the class. A method is a function associated with a class. Most methods operate on objects that-is, instances of a class. Some methods (called **static** methods) operate on class types themselves. An interface represents the **public**ed methods related to a class instance (object).

A **constructor** is a special method that creates (memory allocation) and initializes instanced objects. Although the constructor declaration specifies no return value, when a constructor is called using a class reference (**new**), it returns a reference to the object it creates. A class can have more than one constructor. If the construction fails, the reference contains a NULL reference.

A **destructor** is a special method that destroys the object where it is called and deallocates its memory. When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

The constructor and destructor methods are public methods by default unless there is a static method what calls the constructor by class reference. In that case the constructor with specified signature becomes local. The destructors are called automatically when the object gets out of scope.

A **singleton** class has one - single - instantiation (object). The single constructor (in this case multiples are not allowed) is in fact a static method with no arguments. A singleton object is created using a class reference (**new**), thus the memory allocation is on the heap, never on the stack. After the singleton is created, successive 'new's return a reference to the already created object, increasing a reference counter which is decreased on delete. If the reference counter becomes zero the destructor of the singleton is called.

```
public singleton class <MyClass>
```

Singleton classes could be used - in combination with garbage collection - to support RAII (Resource Acquisition Is Initialization). For example, claiming hardware or file resources.

Inheritance - even multiple inheritance - is supported by declaring the ancestor class within the body of the descendant class. There should be no name clashes. If name clashes occur and when the method starts with a letter the keyword **rename** provides automatic renaming with an underscore prefix.

Inherited classes are private by default. The **public** keyword is explicitly required if descendent classes are to be passed to ancestor class methods. Overloading (different method signature)

ancestor class methods is allowed, however polymorphism is only possible by inheriting **abstract** classes - classes with only method prototypes, no fields. All defined methods within abstract classes are public by default. Abstract classes are not instantiated. Nested classes are not allowed. However, when structs are declared within a class, the class has access to all fields of this structs, also the private fields.

```
abstract class <IMyClass>
  <method>( IN : <IMyClass> ) : BOOL = FALSE
```

Abstract class methods that return values could be set to return default values. If an ancestor class method is not required to be implemented by the descendant class, the method will return the default values if called. The descendant class sets the virtual table position for this ancestor class method with a function pointer entry - which returns the default values. This construction is only possible for ancestor methods where returned values are **all** set to default values, all other methods are mandatory to be implemented by descendant classes.

```
public class <MyClass>
  public <IClass>           // Class <IClass> is inherited and public.
  rename <IClass>.<method>  // rename <method> to <_method>.
  number : U32              // Class U32 is aggregated.
  public <MyClass>.<method> // Now <MyClass>.<method> and
    .                       // <MyClass>.<_method> exist.
```

A class variable implements class aggregation. Declare the class **final** if no class descendants are allowed.

```
public final class <MyClass>
```

Overriding methods like other object oriented programming languages is not supported. SF class method polymorphism is solely supported through abstract classes.

Enums and sets

Enums (enumerations) could be unnamed:

```
enum
{
  RED = 1     // Default assignment is 0, RED is assigned 1.
  BLUE        // BLUE is assigned 2.
  GREEN       // GREEN is assigned 3.
}
```

Enums have default type ENUM. When an **enum** is named, it becomes a new type which inherits from ENUM and the compiler checks for the new type. There are ENUM type methods available: in, size, count, min, max and - if the enumerated values are continuous - range:

```
enum COLOR : U8  // Occupy U8 space if used for variable declarations.
{
  RED   = 1      // COLOR.min = 1.
  BLUE  = 2      // COLOR.max = 3.
  GREEN = 3      // COLOR.range, pushes min and max on data stack.
}

if getColor in COLOR.range  // Method call style -> ENUM.in( getColor(),
                            // COLOR.range()).
{
  .
```

```
}
```

The `'in'`-method is not recognized as an operator, but the infix notation could be allowed. Its signature and implementation is:

```
public overload in( value : U8, min : U8, max : U8 ) : BOOL
# this.allowInfix
# this.highLight( 0xFF0000 )
  return (( value >= min ) and ( value <= max ))

public overload in( value : U8, inSet : U8[ : 2 ] ) : BOOL
# this.allowInfix
# this.highLight( 0xFF0000 )
  index  : U8   = 0
  result : BOOL = TRUE
  while result and ( index < inSet.count )
    result = (( value >= inSet[ index : 0 ] ) and ( value <= inSet[ index : 1 ]
             ))
    inc( index )
  return result
```

Enums could be inherited by other new enums:

```
public enum NEW_COLOR
{
  [ <module>. ]COLOR  // Inherit COLOR, declare NEW_COLOR.RED, -BLUE and -
                      // GREEN.
  YELLOW              // NEW_COLOR.YELLOW assigned 4.
  BROWN = 8           // NEW_COLOR.BROWN assigned 8.
}
```

Enums could be descending:

```
public descending enum ERROR
{
  E1                  // Default assignment is -1.
  E2                  // ERROR.E2 assigned -2.
  E3
  E10 = -10
}
```

Enums could be extended, however this means that count and either the min (descending enum) or max will be declared as a public ENUM variable by the compiler:

```
extend enum [ <module>. ]COLOR
{
  YELLOW              // YELLOW assigned 4.
  BROWN = 8           // BROWN assigned 8.
}
```

There should be no name clashes when inheriting or extending enums.

Enums are a set of named values. **Sets** are dynamic arrays of subranges and inherit from ENUM and are derived from cardinal (enum) types but have no named values itself.

```
public set HEX : CHAR
{
  '0' .. '9'
  'A' .. 'F'
}
```

Structs and unions

A struct represents a heterogeneous set of elements. An union is a struct with overlapping elements. Each element is called a field; the declaration of a struct type specifies a name and type for each field:

```
public struct s_WINDOW
  screenWidth  : U8 = 80   // Initialize on construction.
  screenHeigth : U8 = 25
```

Struct fields are public by default. However, they could be declared private with the keyword **private**, those are not exported. Within the module or class there is still access to the private fields. There is no mechanism to declare union or variant parts as in other programming languages. The rationale for this that if parts of a struct could have a different meaning than it is considered as different types. However, the inherit mechanism allows to declare a common and a different part:

```
struct s_NAME
  firstName : STR255( 20 )
  lastName  : STR255( 40 )

public struct CITIZEN
  s_NAME                   // Inherit struct NAME.
  address : STR255( 40 )

public struct EMPLOYEE
  s_NAME                   // Declare EMPLOYEE.firstName and EMPLOYEE.lastName.
  department : STR255( 30 )

union u_NODE_WORD
  word : U16
  byte : BYTE[ 2 ]
  struct : U16
  // bitfields
  node_6_0    : B7 // node position 6-0 bits
  parity_low  : B1 // Even parity low 6-0 bits (7 bits)
  node_12_7   : B6 // node position 12-7 bits
  ring_dir    : B1 // 0 = clockwise, 1 = counterclockwise
  parity_high : B1 // Even parity high 14-7 bits (7 bits)
```

Struct inheritance - even multiple inheritance - is supported by declaring the ancestor struct within the body of the descendant struct. There should be no name clashes (from fields or structs). Descendent structs could be passed to ancestor struct functions.

Class inheritance and polymorphism could provide the same union or variant semantics keeping type safety. A reference to a superclass could also be used to refer to any subclass.

Declaration of infix operators is not possible for structs. If this is required, declare the struct as a class.

Dependencies

Dependencies are declared through the keyword **use**. Methods or functions could be excluded from classes or modules.

```
use <class>
use <method>
use <class> [ exclude <method> [, <method> ] ]
use <class>.<method> [, <method> ]
```

```
use <module> [ exclude <function> [, <function> ] ]
use <module>.<function> [, <function> ]
```

Stack manipulations

Regarding the stack you should be aware that optimization could cause function parameters or results to be passed through the processor' registers instead of the processor' stack. This should be kept in mind when speaking of 'stack' or 'stack' operations. A function declaration shows what happens on the stack:

```
public show( IN par1 : U32, INOUT par2 : *U32 ) : U32, STATUS
```

The parameters `par1` and `par2` are pushed on the stack by the caller. The order is from left to right: `par1` first, then `par2`. The callee pushes `U32` and `STATUS` on stack before returning to the caller. The same order from left to right: `U32` first, then `STATUS`.

For multiple assignment the same order rules:

```
{
  par1   : U32
  par2   : U32
  par3   : U32    = 0
  status : STATUS = STATUS_OK

  par3, status := show( par1, par2 )
}
```

or in case of result evaluation:

```
{
  // The stack is balanced for each statement unless an unbalanced
  // stack is allowed within a statement block. In that case the
  // compiler encounters a return, push or pop call. However, within
  // loops a balanced stack is mandatory. The use of 'pop' is only
  // possible when there is data on the stack within the scope
  // of 'pop', otherwise a stack underflow error will be given
  // (compile time check)!
  .
  .

  // par3 will be assigned when STATUS result matches OK. The par3
  // result will be lost (popped from stack) when the STATUS result
  // does not match OK.
  if ( par3 := show( par1, par2 )) = STATUS_OK
  { // Data (status) is consumed.
    .
  } // Data (par3) popped from stack if not STATUS_OK.

  if ( par3 := show( par1, par2 )) = STATUS_OK
  {
    .
  }
  else
  {
    pop par3           // Data is now popped from stack into par3.
  }

  .
  push par3            // Pushed on return stack.
  .
```

```
                        // Optional statements to handle other things.
}
```

The keyword **return** returns the program execution to the caller. Parameters passed with return are pushed on the return stack. The actual 'return' happens when return becomes out of scope and optional statements after return are handled and destructions are carried out.

`'INOUT'` and `'OUT'` parameters passed to a function are not allowed to return and all passed parameters are not allowed access after the keyword return if return passes results on the return stack. Function parameters and returned results share the same stack area! Only `'IN'` parameters and local variables could be returned. The keywords `push <var>` and `pop [ var ]` could be used to manipulate the data stack.

Contract

The **pre**, **post** and class **invariant** statements provide design by contract. Like exception handling **pre**, **post** and **invariant** should not be abused to replace the functionality of returning errors, statuses or booleans. If a **pre**, **post** or **invariant** condition fails an exception is raised. This implies run-time overhead. The **pre** statement is evaluated on entry, while the **post** statement is evaluated on exit of a function or method. The class **invariant** is evaluated on write access of its fields.

Control structures

The definitions **TRUE** and **FALSE** are reserved SF keywords. **FALSE** is a constant which equals zero (**0**). **TRUE** for assignments equals minus 1 (**-1**), but as a condition it is <> **FALSE**.

Statements within square brackets [] are optional.

```
   if <logical statement or function with BOOL result>
    .
[ else if <logical statement or function with BOOL result> ]
    .
[ else ]
    .
```

Ternary conditional operator. If <condition> is true then `a := b` else `a := c:`

```
a := <condition> ? b : c
```

```
while <logical statement or function with BOOL result>
  .
[ break ]         // Leave loop.
[ continue ]      // Start next iteration.
  .

do
  .
[ break ]
[ continue ]
  .
while <logical statement or function with BOOL result>

for <initialize variable> to <stop value variable>
  .
[ break ]
[ continue ]
[ retry ]         // Start again.
```

```
  .
  <variable>.<operator>    // Compiler expects operator on variable for
                           // next iteration.

loop <variable or value>   // Iterate <variable or value> times.
  .
[ break ]
[ continue ]
[ retry ]
  .

loop              // Loop forever.
  .
[ break ]
[ retry ]
  .

case <variable>
  <select> : <statement>   // <select> is valid value or set for variable.
    .
  [ break ]
  <select> : <statement>
  <select> : <statement>
  [ break ]
  default
    <statement>
```

Algebraic assembly

Assembly is microprocessor specific and is initiated by the keyword **asm**. The assembly module - for generating machine code - is declared by **use**. Inside the grouped assembly statements the scope is locked for the assembly module. Outside function or method calling inside the **asm** scope is only possible in case of native code generation. Other references could be passed but never the values hold by those outside references. Assembly has its own operators and control structures.

Exception handling

Exception handling should not be abused to replace the functionality of returning errors, statuses or booleans. Raising exceptions causes run-time overhead, because generally at the **raise** statement a stack dump is generated. Exception handling is done with the **try-catch** statement:

```
try
  .
  .
catch
[ raise <error number> [, parameter ] ]
[ reraise ]
[ retry ]
[ return [ <result> ] [, <result> ] ]  // Ignore exception, leave.
```

Every thread - and there is at least one - has a catch pointer holding the location where the program execution jumps to when an exception has been raised. The **try** statement stores the thread' catch pointer in a local catch pointer and replaces the thread' catch pointer with its own matching catch pointer. The first thing the **catch** statement does is to restore the local catch pointer in the thread' catch pointer. The **raise** statement sets the last error of the thread, pushes parameters on the data stack if required, generates a stack dump and jumps to the thread' catch pointer location. The stack dump is only generated when there is a way to retrieve it. The **reraise** statement just jumps to the

thread' catch pointer location, while the **retry** statement jumps to the matching **try** position where again the thread' catch pointer is temporary stored and replaced. Exceptions could be raised by hardware interrupts e.g. divide-by-zero, invalid-memory-access etc.

Templates

Templates are a mechanism to implement the concept of genericity. The **template** keyword tells the SF compiler that the function or class definition that follows will manipulate one or more unspecified types or values. At the time the actual function class code is generated from the template, those types or values must be specified so that the compiler can substitute them. The substitution requires that specified classes and their operators are defined.

```
template <T> gt( in1 : T, in2 : T ) : T
  return ( in1 > in2 ) ? in1 : in2    // Requires operator T.> defined.
```

For functions templates only the signature could contain unspecified types or values. The mechanism is similar to overloading functions.

```
template <T, N [: U16]> class Stack
  T data[ N ]  // Fixed capacity is N elements.
  .
```

The type specifier for the value N is optional. For classes the template actually creates a new class type when implemented: e.g. Stack<U8,100>.

Concurrency

The keyword **process** defines concurrency of code. Optionally a process name and priority could be defined. In case of a processor target the stack required for every process is exactly known because all source code is evaluated (global analysis) and no recursion is allowed. Therefore no stack size parameter has to be given, no stack evaluation is necessary and data stack overflow is simply not possible. All dynamic memory allocation is done on the heap.

Parameters within square brackets [] are optional.

```
type PRI8 = U8 // Priority type P8.
process [name] [ ( [<sensivity list>,] [enable : BOOL = TRUE,] [priority : PRI8
                = 0] ) ]
  .
  .
```

Example:

```
object SpiSlave(
# param CPOL = 1  // CPOL = 0, first clock edge = 0/1, second edge = 1/0
transition
                  // CPOL = 1, first clock edge = 1/0, second edge = 0/1
transition
# param CPHA = 1  // CPHA = 0, data valid during first clock edge
                  // CPHA = 1, data valid during second clock edge
# param SPI_DATA_WIDTH = 8
    IN  clk : B1,
    IN  spi_sck : B1,
    IN  spi_mosi : B1, // Master out, slave in
    OUT spi_miso : B1, // Master in, slave out
    IN  spi_ssel_n : B1,
    IN  spi_m2s : B[ SPI_DATA_WIDTH ], // Data from master
```

```
    IN  spi_m2s_valid : B1
    OUT spi_s2m [ SPI_DATA_WIDTH ], // Data from slave
    OUT spi_s2m_valid : B1,
    )
    spi_active : B1
{
     .
     .
  process synchronize( PE clk )
    i_sck <= spi_sck
    i_ssel_n <= spi_ssel_n
    i_mosi <= spi_mosi

  process shift_and_count( PE clk )
  {
    .
    .
  }
}
```

Processes could be **suspend**ed or **resume**d during running by means of the enable parameter and the sensivity list. If the sensivity list is empty, changes in the process inputs are considered as process activation. Inputs could be parameter prefixed in the sensivity list with the kind of activation `'PE'` positive edge or `'NE'` negative edge.

In case of a processor target communication between processes (IPC - inter process communication) is possible through **Channel**s (unbuffered messages), **Queue**s (buffered messages) or **Socket**s (remote buffered messages). The keywords **open** (constructor), **close** (destructor), assignment (:=, <=) or **read** and **write** give access to the messages. Remote buffered messages require handshaking through **connect** and **accept**.

Memory management

Dynamic memory is allocated by **new** (construction). Reference smart counters keep track of assignments, parameter passing, reference out of scope and deletion by **delete** only when required.

All dynamic memory allocation is automatically destroyed (destruction) when the reference counter becomes zero. Assignments and parameter passing increment the reference counter. Assignment could also decrement the reference counter when the reference itself is overwritten. Out of scope (also as stack parameter) and **delete** decrements the reference counter. The actual destruction happens when the reference counter equals zero or could be delayed when the garbage collection is active. If there is a single point of destruction (by delete or out of scope) on a single thread the reference counter is set to one on memory allocation and not incremented during its life cycle.

Memory leaks are not possible, circular references are not allowed. The compiler gives an error when a circular reference is found. Recursion is not allowed except for object functions. Because of global analysis the required stack space is known for every part of the program. Stack overflows or underflows are not possible. Boundaries of (dynamic) arrays are checked if required.

```
{
  .
  array100U8 : U8[ ] = createAndFillU8Array( 100 )
  .                                       // Ref. counter increments (=1).
}                                         // Out of scope. Ref. counter
                                          // decrements (=0). Destruction.
```

```
createAndFillU8Array( IN size : U32 )
{
  allocatedArray : U8[ ] = new U8[ size ]
                                          // Ref. counter increments (=1).
  .
  fillArray( allocatedArray )             // Ref. counter increments (=2).
  .                                       // Ref. counter decremented (=1).
  return allocatedArray
}

fillArray( IN dynamicArray : U8[ ] )
{
  .
  checkArraySize( dynamicArray )          // Ref. counter increments (=3).
  .                                       // Ref. counter decremented (=2) by
  .                                       // exit checkArraySize().
}                                         // Out of scope dynamicArray.
                                          // Ref. counter decrements (=1).

{
  allocatedArray : U8[ ] = new U8[ 100 ] // Ref. counter increments (=1).
  .
  allocatedArray = new U8[ 100 ]          // Old reference overwritten. Old
  .                                       // ref. counter decrements (=0).
  .                                       // Destruction.
  .                                       // New ref. counter increments
  .                                       // (=1).
  .
  pDynamicArray : *U8[ ] = allocatedArray
  .                                       // New ref. counter increments
  .                                       // (=2).
  delete allocatedArray                   // Deletion. New ref. counter
  .                                       // decrements (=1).
}                                         // Out of scope pDynamicArray. New
                                          // ref. counter decrements (=0).
                                          // Destruction.
```

Garbage collection

Garbage collection is possible when multi-threading is supported. Garbage collection is nothing more than a process with the lowest priority what runs when all other processes are idle or it is a process with a (low) fixed partition. It takes care for the actual destruction of allocated dynamic memory from references with zero counts. In case of priority based process scheduling the priority of the garbage collection process inverses when higher priority processes require memory allocation what is not available yet.