OTL Tech Note - Injection of Complex Data Classes

Charles Rivet

Table of Contents

Injection of Complex Data Classes

Category | Purpose | Intended Audience | Applicable to | Description | Limitations | See also

Category:

Top

Frequently Asked Questions

Purpose:

Top

To provide a guideline and a working example demonstrating how to inject complex data classes based on RTSequence, RTSequenceOf, and RTByteBlock.

Intended Audience:

Top

ObjecTime developers, testers, and users.

Applicable to:

Top

ObjecTime 5.1 and up

Description:

Top

A common question asked concerns the use of RPL to inject complex data classes into a daemon.

This document assumes that the reader is familiar with the ObjecTime RTS Control Panel and the creation of daemons. This technical note will only deal with the RPL code necessary to inject a data class based on RTSequence or RTSequenceOf, or an RTByteBlock.

The following files provide the working example for this technical note. The explanations that follow are all based on the classes defined in those files.

- injectionTechnoteV*n*.update
  This is the ObjecTime Developer 5.1 update file, where "*n*" is a number representing the current version.

- injectionTechnoteV*n*.requpdate
  This is the ObjecTime Developer 5.1 requirements update file, where "*n*" is a number representing the current version.

- injection.h
  This file contains the definition for a simple external data class that is used to demonstrate RTByteBlock.

- injection.cpp
  This file contains the code of the methods defined in injection.h. To compile, execute the following commands:
  On Windows NT: cl /c /Zl /Tp injection.cc
  On Sun Solaris: g++ -c injection.cc

- ShowStore.c
  This file, when compiled and linked, generates an executable that helps determine the contents of an RTByteBlock injection sequence. To compile and link, execute the following commands:
  On Windows NT: cl ShowStore.c
  On Sun Solaris: gcc -o ShowStore ShowStore.c

Note that it may be necessary to re-define the location of the "injection" external data package to point to the correct directory and object (.obj or .o) file. Or it may be simpler to delete the package and merge it again by dragging the injection.h file from the directory browser to the package pane of the update browser and adding the Sinject class to the CPP_Example package.

RTSequence

An RTSequence is the equivalent to a C++ class that contains other classes. There are two cases that we need to examine here: all data members are public and some (or all) data members are private.

All Data Members Public

This is the simplest case. In this case, every data member can be accessed directly by reference. Let us first define the sequence with which we will work in this example:

```
DATA CLASS Csequence
   ISASEQUENCE
   FIELDS
      {
      DEFINE f_integer ISA PUBLIC Integer DEFAULT '1';
      DEFINE f_boolean ISA PUBLIC Boolean DEFAULT 'true';
      DEFINE f_character ISA PUBLIC Character DEFAULT '$a';
      DEFINE f_real ISA PUBLIC Real DEFAULT '1.5';
      DEFINE f_string ISA PUBLIC String DEFAULT '"The quick brown fox"';
      } /* end of fields */
   ; /* end of CSequence */
```

To initialize such a structure in RPL, one only has to do the following:

```
| |
SEQUENCE CSequence FIELDS
   f_integer: 128.
   f_boolean: false.
   f_character: $S.
   f_real: 3.1415926536.
   f_string: 'jumped over the lazy dog. '
ENDSEQUENCE
```

Some Data Members Private

This is a bit more complex. In this case, at least one data member is private and must be accessed through an accessor function. Note that this can easily be extended to all data members being private. The only restriction here is that there must be a method provided to set the private data member.

Let us then define the sequence with which we will work in this example:

```
DATA CLASS CsequencePriv
```

```
   ISA SEQUENCE
   FIELDS
      {
      DEFINE f_integer ISA PUBLIC Integer DEFAULT '1';
      DEFINE f_boolean ISA PUBLIC Boolean DEFAULT 'true';
      DEFINE f_character ISA PUBLIC Character DEFAULT '$a';
      DEFINE f_real ISA PUBLIC Real DEFAULT '1.5';
      DEFINE f_string ISA String DEFAULT '"The quick brown fox"';
      } /* end of fields */
   INSTANCE METHODS
      {
      DEFINE SetString LANGUAGE 'RPL'
         CODE
            {
            SetString: aString
            | |
            f_string := aString
            };
      DEFINE SetString LANGUAGE 'C++'
         ATTRIBUTES {Inheritable Polymorphic Public}
         TYPE 'void' ARGUMENTS 'String aString'
         CODE
            {
            f_string = aString;

            };
      } /* end of instance methods */
   ; /* end of CSequencePriv */
```

We can see that the only differences between this data class and the previous one are that "*f_string*" is now private and that methods have been defined to set this string.

To initialize such a structure in RPL, one only has to do the following:

```
| tmpStruct |
tmpStruct := SEQUENCE CSequencePriv
   FIELDS
      f_integer: 100.
      f_boolean: false.
      f_character: $S.
      f_real: 123.456
   ENDSEQUENCE.
TmpStruct SetString: 'This is a (private) string (really)!'
^ tmpStruct
```

Note that we now have to define a local variable, "*tmpStruct*", in order to make the method call to initialize the private data member.

RTSequenceOf

An RTSequence is the equivalent to a C++ array. It can be an array of an internal ObjecTime data type, such as RTInteger, RTReal, or RTString, or of a class derived from RTSequence.

Let's start with the simplest case: a sequence of base classes.

RTSequenceOf Basic Type

In the present context, I define "basic type" to mean any of the "standard" ObjecTime data classes: (RT)Boolean, (RT)Character, (RT)Integer, (RT)Real, and (RT)String. The following class defines a RTSequenceOf Integer, but it could be any of the above types.

```
CONSTANT kSequenceSize ISA Integer VALUE '5';
DATA CLASS CSequenceOf
   ISA SEQUENCEOF [kSequenceSize] TYPE Integer
   ; /* end of CSequenceOf */
```

To initialize such a structure in RPL, one only has to do the following:

```
| tempArray |
tempArray := CsequenceOf with: 51 with: 52 with: 53 with: 54 with: 55.
^ tempArray
```

Again, we have had to define a local variable that is created as a RTSequenceOf with the correct number of initializing "with:" statements. Note that although a constant can (and should) be used to define the size of the RTSequenceOf, the tester **must** be aware of this value in order for this to work!

RTSequenceOf Complex Type

In the present context, I define "complex type" to mean any class derived from RTSequence. The following class defines a RTSequenceOf Csequence, as defined in the previous section.

```
DATA CLASS CSequenceOfSequence
   ISA SEQUENCEOF [kSequenceSize] TYPE CSequence
   ; /* end of CSequenceOfSequence */
```

As we have previously seen how to initialize a simple RTSequenceOf and a RTSequence, it is simply a matter of combining the two techniques. To initialize such a structure in RPL, one then only has to do the following:

```
| tempSeqOfSeq |
tempSeqOfSeq := CSequenceOfSequence new: kSequenceSize.
FOR index IN 1 to: tempSeqOfSeq size DO
   tempSeqOfSeq
      at: index
      put:
         SEQUENCE CSequence FIELDS
            f_integer: kSequenceSize - index.
            f_boolean: false.
            f_character: $r.
            f_real: 987.654.
            f_string: 'Injecting...'
         ENDSEQUENCE
ENDFOR.
^ tempSeqOfSeq
```

Again, we have had to define a local variable that is created as a CSequenceOfSequence with a size of kSequenceSize. Note that in this case, the constant previously defined can be used as we are now going through a loop to initialize the data. The same technique could have been used previously if all the values were to be the same. Similarly, if in this case also all the values are different, one could use the following code:

```
| tempSeqOfSeq |
tempSeqOfSeq := CSequenceOfSequence new: kSequenceSize.
TempSeqOfSeq at: 1
   put:
      SEQUENCE CSequence FIELDS
         f_integer: 1.
         f_boolean: false.
         f_character: $A.
         f_real: 1.1.
         f_string: 'Injecting #1...'
```

```
      ENDSEQUENCE
...
TempSeqOfSeq at: 5
   put:
      SEQUENCE Csequence FIELDS
         f_integer: 5.
         f_boolean: true.
         f_character: $E.
         f_real: 1.5.
         f_string: 'Injecting #5...'
      ENDSEQUENCE
^ tempSeqOfSeq
```

Naturally, this can be a pain if kSequenceSize is large...

RTSequence of RTSequenceOf

This is one type that I intentionally skipped during the previous discussion on RTSequence as I wanted to deal with the initialization of RTSequenceOf first. As with the RTSequenceOf of RTSequence, it is simply a matter of combining the techniques demonstrated previously. In order to do this, we define the following class:

```
DATA CLASS CsequenceSequenceOf
   ISA SEQUENCE
   FIELDS
      {
      DEFINE f_sequenceof ISA PUBLIC CSequenceOf;
      DEFINE f_otherSequenceof ISA PUBLIC CSequenceOf;
      } /* end of fields */
   ; /* end of CsequenceSequenceOf */
```

We could define more than one field for the sequence, but it is sufficiently to demonstrate with only one.

To initialize such a structure in RPL, one only has to do the following:

```
| tempArray |
tempArray := CSequenceOf with: 31 with: 32 with: 33 with: 34 with: 35.
SEQUENCE CSequenceSequenceOf FIELDS
   f_sequenceof: tempArray.
   f_otherSequenceof: tempArray.
ENDSEQUENCE
```

RTByteBlock

We finally come to the dreaded RTByteBlock. The initialization of an RTByteBlock can be simplified a lot if you only think of it as a sequence of bytes. We can then use the same techniques used in the RTSequenceOf section to initialize a RTByteBlock.

There are, however, some important caveats!

If this RTByteBlock is used to represent an external data type (as is being done in the example available with this technical note), you will need to know and understand the data structure of that external data type. In addition, you will have to understand the storage mechanism used by the compiler (byte/word/long alignment) and the target microprocessor (little/big-endian). This problem is further compounded by the use of different microprocessors for simulation, emulation, and target as you will then potentially need multiple initialization scheme.

So let's define the following external data type (found in injection.h):

```
class SInject
{
   public:
      SInject();
      SInject( SInject * pSInject );
      SInject( int iInt, float fFloat, char cChar );

      void    SetInteger( int iInt );
      void    SetFloat( float fFloat );
      void    SetChar( char cChar );

      int       GetInteger();
      float    GetFloar();
      char    GetChar();

      char *    printString();

   private:
      int       l_integer;
      float    l_float;
      char    l_char;
};
```

When using either the Microsoft Visual-C++ 5.0 (NT) or the GNU C++ 2.7.1 (Sun Solaris) compiler, this results in a 12 bytes structure because of the default alignment on 4-byte boundaries. In this structure, bytes 14 (03) represent "*l_integer*", bytes 58 (47) represent "*l_float*", and bytes 912 (811) represent "*l_char*" (although only byte 9 (8) is significant).

If such a structure is to be transferred using a RTByteBlock, one could then initialize the RTByteBlock using something like:

On little-endian systems: *(e.g., Windows NT)*	**On big-endian systems:** *(e.g., Sun Solaris)*
| byteStream | byteStream := RTByteBlock new: 12. byteStream at: 1 put: 51. byteStream at: 2 put: 1. byteStream at: 3 put: 0. byteStream at: 4 put: 0. byteStream at: 5 put: 0. byteStream at: 6 put: 1. byteStream at: 7 put: 128. byteStream at: 8 put: 64. byteStream at: 9 put: 98. byteStream at: 10 put: 0. byteStream at: 11 put: 0. byteStream at: 12 put: 0. ^ byteStream	| byteStream | byteStream := RTByteBlock new: 12. byteStream at: 1 put: 0. byteStream at: 2 put: 0. byteStream at: 3 put: 1. byteStream at: 4 put: 51. byteStream at: 5 put: 64. byteStream at: 6 put: 128. byteStream at: 7 put: 1. byteStream at: 8 put: 0. byteStream at: 9 put: 98. byteStream at: 10 put: 0. byteStream at: 11 put: 0. byteStream at: 12 put: 0. ^ byteStream

This would result in:

l_integer = 307   (0x133)
l_float = 4.000122
l_char = b

Note 1:*It is not necessary to "put" a value of zero (0) at NT bytes 3, 4, 5, 10, 11, and 12 or at Sun Solaris bytes 1, 2, 8, 10, 11, and 12 as the byteStream structure is initialized to zeros (0's) on creation. They are, however, included for completeness.*

Note 2:*Do not forget to use the* "BLOCK_OF"*macro when decoding a RTByteBlock...*

Whenever we deal with an RTByteBlock, and as can be seen from the previous example, it is evidently very important to know the representation of the data in memory. The following sections explain this for the integer and float data types.

Integer Internal Representation

Microsoft Windows NT uses a little-endian data storage scheme, i.e., an integer is represented as a series of bits from the least to the most significant (LSBtoMSB). That is why for an integer value of 307 (0x133), in the above example, the first byte is initialized to 51(0x33), the second byte to 1(0x1), and the last two bytes to 0 (0x0). This gives the following binary representation of this data:

51=0x33 1=0x01 0=0x00 0=0x00
11001100 10000000 00000000 00000000

If the target operating system uses a big-endian data storage scheme (MSBtoLSB), such as is the case with Sun Solaris and most UNIXes, then the first two bytes would be initialized to 0, the third byte to 1, and the fourth byte to 51. This would then give the following binary representation:

0=0x00 0=0x00 1=0x01 51=0x33
00000000 00000000 00000001 00110011

Note that although the value of each byte is the same (ether in decimal or hexadecimal notation), their binary representations are different.

Note that negative numbers are represented using the two's complement scheme. To get the internal storage representation of a negative integer, take the binary representation of the number's absolute (i.e., positive) value, flip all the bits (i.e., 1 becomes 0 and 0 becomes 1), and add 1 to the result.

Real (floating point) Internal Representation

Under both Microsoft's Windows NT and Sun Solaris, a floating point value is represented using the IEEE numeric standards for the C/C++ float (4-byte real) and double (8-byte real) floating point types. Microsoft also defines a "long double" type for 10-byte real values. As this is a non-standard value, it will not be discussed here. However, the same little/big-endian rules, as seen for integers, apply here too.

Floating point values are stored in three segments: Sign (S), Mantissa (M), and Exponent (X). These three segments relate in the following manner:

Floating point value = S * M * 2^X.

The sign is always stored as a single bit. A value of zero (0) indicates a positive number and S=1 in the above equation. A value of 1 indicates a negative number and S=-1 in the above equation.

The mantissa and exponent are represented by a different number of bits depending on the number of bytes used for the floating point number:

float: sign bit 8-bit exponent 23-bit mantissa
double: sign bit 11-bit exponent 52-bit mantissa

The mantissa is stored as a binary fraction of the form 1.xxx. This fraction has a value greater than or equal to 1 and less than 2. Note that real numbers are always stored in normalized form, that is, the mantissa is left-shifted such that the high-order bit of the mantissa is always 1. Because this bit is always 1, it is assumed (not stored) in 4- and 8-byte floating point formats, so the mantissas are actually 24 or 53 bits, even though only 23 or 52 bits are stored. The binary (not decimal) point is assumed to be just to the right of the leading 1.

The exponents are biased by half of their possible value. To get the actual exponent, you subtract this bias from the stored exponent. If the stored exponent is less than the bias, it is then a negative exponent. The exponents are biased as follows:

float: 8-bit exponents are biased by 127
double: 11-bit exponents are biased by 1023

The formats for the various sizes is then as follows (Byte #1 is the last byte on Windows NT and the first on Sun Solaris):

```
      Byte #1   Byte #2   Byte #3   Byte #4   ... Byte #8
float:  SXXX XXXX XMMM MMMM MMMM MMMM MMMM MMMM
double: SXXX XXXX XXXX MMMM MMMM MMMM MMMM MMMM ... MMMM MMMM
```

Demo Software for Internal Representation

In order to help understand the internal representation of integer and floating point numbers, and with the translation of simple data types into RTByteBlock injection sequence, a small C program can be obtained. This program takes an integer (int or long) or a real (float or double) and displayed its internal structure along with the required injection sequence. This program can be created by compiling and linking ShowStore.c. It can the be used with the following syntax:

```
ShowStore -[i|l|f|d] number
```

This program works in the same way on Windows NT and on Sun Solaris.

How to Use the Model Example

Now that we have seen how to do it, I will spend a bit of time explaining the model built to illustrate this. The model is intentionally simple as it only needs to show the principles mentioned above. It does, however, also provide examples of complex data initialization in code as well as in inject messages.

The model consists of three actor classes: Sender, Receiver, and Container. The model is also separated into two main packages: *CPP_Example* and *RPL_Example*. Each package respectively contains C++ and RPL versions of the three actors. The model also consists of two protocol classes (*DataProtocol* and *ControlProtocol*) and various data classes as described previously.

All initialization operations and received data are logged to the system log for easy observation of the results.

Protocol Classes

The *DataProtocol* class is used to exchange information between the sender and receiver actors. This is the protocol into which complex data will be injected. The definition of this protocol is as follows:

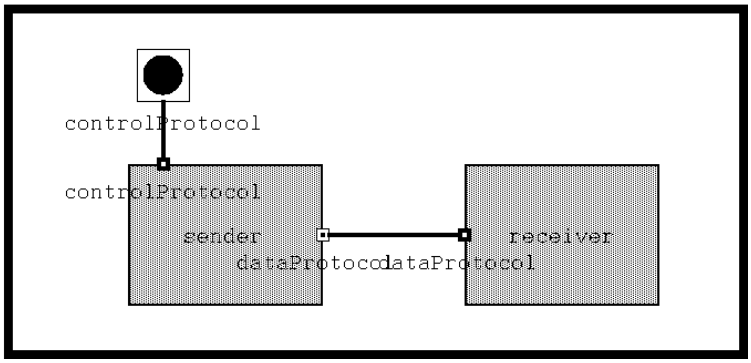
```
PROTOCOL CLASS DataProtocol
   SERVICE BasicCommunication
   IN MESSAGES
      {
      DEFINE SigSeq ISA CSequence;
      DEFINE SigSeqByteBlock ISA CsequenceByteBlock;
      DEFINE SigSeqOf ISA CSequenceOf;
      DEFINE SigSeqOfByteBlock ISA CsequenceOfByteBlock;
      DEFINE SigSeqOfSeq ISA CsequenceOfSequence;
      DEFINE SigSeqSeqOf ISA CsequenceSequenceOf;
      DEFINE SigByteBlock ISA RTByteBlock;
      DEFINE SigSeqPriv ISA CsequencePriv;
      } /* end of in messages */
   ; /* end of DataProtocol */
```

The *ControlProtocol* class is used by the sender actor to send predefined messages that are initialized within that actor. It was only defined to check for any differences between internal initialization and sending of data and sending of data through the injection of a message. The definition of this protocol is as follows:

```
PROTOCOL CLASS ControlProtocol
   SERVICE BasicCommunication
   IN MESSAGES
      {
      DEFINE SendSeq ISA Null;
      DEFINE SendSeqOf ISA Null;
      DEFINE SendSeqByteBlock ISA Null;
      DEFINE SendSeqOfByteBlock ISA Null;
      DEFINE SendSeqOfSeq ISA Null;
      DEFINE SendSeqSeqOf ISA Null;
      DEFINE SendByteBlock ISA Null;
      DEFINE SendSeqPriv ISA Null;
      } /* end of in messages */
   ; /* end of ControlProtocol */
```

Container Actor

The container actor's only function is to provide a "shell" for the other two actors. There are two flavours of the container actor: *Container_CPP* for the C++ version of the model, and *Container_RPL* for the RPL version. The following diagram shows the contents of the container actor.



As you can see, the sender and receiver communicate with each other, within the container actor, through the *dataProtocol* port, which is of *DataProtocol* class.

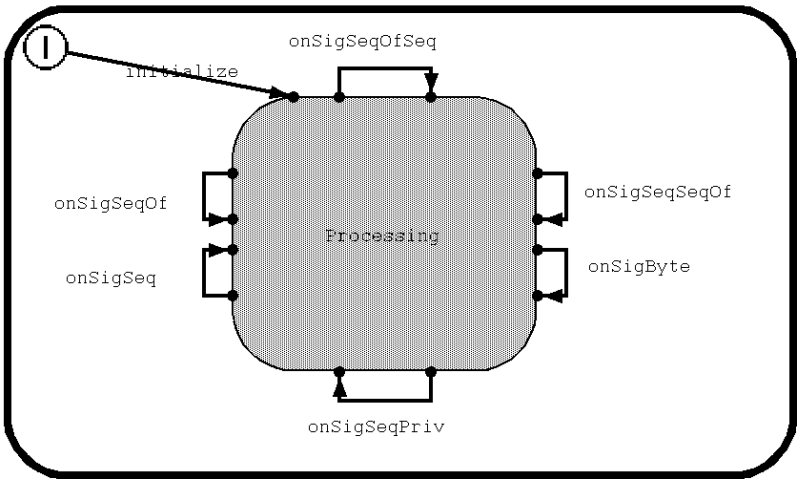
The sender actor also has the capability to send predefined messages when instructed through the *controlProtocol* port, which is of "*ControlProtocol*" class.

Sender Actor

The sender actor is available in two "flavours": *Sender_CPP* is coded in C++ and *Sender_RPL* is coded in RPL.

The sender actor has a single state that handles the different control messages. On this state, a transition is defined for each signal from the *controlProtocol* port. When one of these transitions is taken, the corresponding data is sent through the *dataProtocol* port. The data to be sent is defined in the "*initialize*" transition.

The following diagram shows the state diagram for this actor:



As previously mentioned the "initialize" transition initializes the actor's extended state variables with the information to be sent. In C++:

```
int iNdx;
CSequenceOf tempArray;

for ( iNdx = 0; iNdx < kSequenceSize; iNdx++ )
{
tempArray[iNdx] = iNdx+ 1;
}

log.log("Initializing data...");

//-------------------------------------------------
// Initialize RTSequence
// ---------------------
lSeq.f_integer = 17;
lSeq.f_boolean = 0;
lSeq.f_character = 'z';
lSeq.f_real = 3.1415926535;
lSeq.f_string = "jumped over the lazy dog";
log.cr(); log.log("RTSequence initialized to: ");
log.show(lSeq.printString()); log.cr();

//-------------------------------------------------
// Initialize RTSequencePriv
// -------------------------
lSeqPriv.f_integer = 23;
lSeqPriv.f_boolean = 0;
lSeqPriv.f_character = 'Y';
lSeqPriv.f_real = 6.283185307;
lSeqPriv.SetString( "This is hidden..." );
log.cr(); log.log("RTSequence (priv) initialized to: ");
log.show(lSeqPriv.printString()); log.cr();

//-------------------------------------------------
// Initialize RTSequenceOf
// -----------------------
for ( iNdx = 0; iNdx < kSequenceSize; iNdx++ )
{
lSeqOf[iNdx] = iNdx +1;
```

```
};
log.cr(); log.log("RTSequenceOf initialized to: ");
log.show(lSeqOf.printString()); log.cr();

//-----------------------------------------------
// Initialize RTSequenceOf RTSequence
// ---------------------------------
for (iNdx = 0; iNdx < kSequenceSize; iNdx++ )
{
lSeqOfSeq[iNdx].f_integer = iNdx + 11;
lSeqOfSeq[iNdx].f_boolean = 0;
lSeqOfSeq[iNdx].f_character = 'z';
lSeqOfSeq[iNdx].f_real = 2.71;
lSeqOfSeq[iNdx].f_string = "Sleeping...";
};
log.cr(); log.log("RTSequenceOf RTSequence initialized to: ");
log.show(lSeqOfSeq.printString()); log.cr();

//-----------------------------------------------
// Initialize RTSequence of RTSequenceOf
// ------------------------------------
for ( iNdx = 0; iNdx < kSequenceSize; iNdx++ )
{
   lSeqSeqOf.f_sequenceof[iNdx] = iNdx + 21;
   lSeqSeqOf.f_otherSequenceof[iNdx] = iNdx + 31;
}
log.cr(); log.log("RTSequence of RTSequenceOf initialized to: ");
log.show(lSeqSeqOf.printString()); log.cr();

//-----------------------------------------------
// Initialize external class variable
// ---------------------------------
lSInject.SetInteger( 1 );
lSInject.SetFloat( 1.0 );
lSInject.SetChar( 'a' );

log.cr(); log.log("External class variable initialized to: ");
log.show(lSInject.printString()); log.cr();

log.log("Initialization complete!");
```

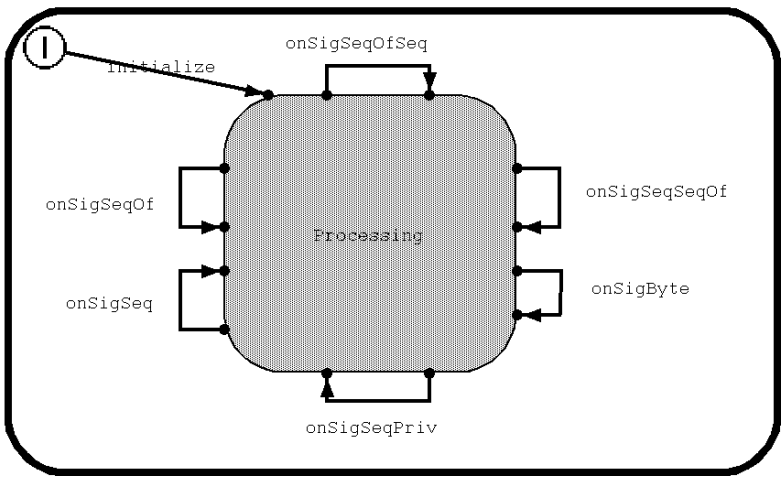
In RPL:

```
| tempArray |
tempArray := CsequenceOf with: 1 with: 2 with: 3 with: 4 with: 5.
log log: 'Initializing data...'.
"--------------------------------".
"Initialize RTSequence".
lSeq := SEQUENCE CSequence FIELDS
f_integer: 17.
f_boolean: false.
f_character: $z.
f_real: 3.1415926535.
f_string: 'jumped over the lazy dog'
ENDSEQUENCE.
log log: 'RTSequence initialized to: ', lSeq printString.
"--------------------------------".
"Initialize RTSequence (Priv)".
lSeqPriv := SEQUENCE CSequencePriv FIELDS
f_integer: 23.
f_boolean: false.
```

```
f_character: $Y.
f_real: 6.283185307
ENDSEQUENCE.
lSeqPriv SetString: 'This is hidden...'.
log log: 'RTSequence (Priv) initialized to: ', lSeqPriv printString.
"--------------------------------".
"Initialize RTSequenceOf".
lSeqOf := CsequenceOf new: kSequenceSize.
FOR index IN 1 to: lSeqOf size DO
LSeqOf at: index put: index
ENDFOR.
log log: 'RTSequenceOf initialized to: ', lSeqOf printString.
"--------------------------------".
"Initialize RTSequenceOf RTSequence".
lSeqOfSeq := CsequenceOfSequence new: kSequenceSize.
FOR index IN 1 to: lSeqOfSeq size DO
LseqOfSeq
at: index
put:
SEQUENCE CSequence FIELDS
F_integer: index.
F_boolean: false.
F_character: $z.
F_real: 2.71.
F_string: 'Sleeping...'
ENDSEQUENCE
ENDFOR.
log log: 'RTSequenceOf RTSequence initialized to: ', lSeqOfSeq printString.
"--------------------------------".
"Initialize RTSequence of RTSequenceOf".
lSeqSeqOf := SEQUENCE CSequenceSequenceOf
FIELDS
f_sequenceof: tempArray
ENDSEQUENCE.
log log: 'RTSequence of RTSequenceOf initialized to: ', lSeqSeqOf printString.
log log: 'Initialization complete!'
```

Receiver Actor

The receiver actor has a state diagram very similar to that of the sender actor:



Each transition on the "Processing" state simply logs the received data to the System log. The only "special" transition is the one for the RTByteBlock: *onSigByte*. In this transition, not only is the message data displayed in its raw form, but it is also displayed as the correct data type associated with the RTByteBlock, *SInject* in this case.

Using the model

Either the RPL or C++ version of the model can be used. However, the RTByteBlock portion of the example has only been implemented in the C++ version in order to get the benefit of the external data class.

To use the RPL model, you just need to compile and load it by right-clicking on the *Container_RPL* actor in the *RPL_Example* package and selecting the "Compile..." option (or pressing "♦C"). In the resulting window, select "Load" and press OK.

To use the C++ model, you will first need to compile the external data class. To do this, open an ObjecTime Developer Command Prompt window, go to the directory that contains the update, and run the injection.bat batch file. This will generate the required object file, if your C++ compiler is setup properly. Once this is done, you will have to modify the "injection" package to point to the correct location, or simply re-create the package by deleting it from the update and then dragging the "injection.h" file from a directory browser window into the package pane of the update browser. You can then compile and load the model by right-clicking on the *Container_CPP* actor in the *CPP_Example* package and selecting the "Compile..." option (or pressing "♦C"). In the resulting window, select "Load" and press OK.

Once the RTS window is up, you will notice that two daemons are already defined: *controlProtocolProbe* and *dataProtocolProbe*. The first daemon, *controlProtocolProbe*, contains the injection messages for the *controlProtocol* port of the sender actor. As such, the injection messages are rather uninteresting.

The second daemon, *dataProtocolProbe*, contains the injection messages for the *dataProtocol* port of the receiver actor. This is the one that contains all the code described in the first part of this technical note and that is probably of the most interest to you.

Finally, before running the model and starting to inject messages, it is suggested that the System log window be open. This will enable you to see all the initialization messages for the sender actor along, and more importantly, with the log messages of all the data received by the receiver actor.

If you desire to see the actual messages, you can also open a trace window on the *dataProtocolPort* daemon.

Note on Testing

If testing requires the injection of many complex data types, it may be beneficial to create a test harness (and/or test harness actors) to do the testing instead of relying on message injection. This would provide for better automation of the process.

Limitations:

Top

The RTByteBlock example is limited to the Microsoft Visual-C++ compiler with an Intel target microprocessor.

See also:

Top

None.