PacketEncoding
IEC 61131 Library for ACSELERATOR RTAC® Projects

SEL Automation Controllers
Table of Contents

Section 1: PacketEncoding
  Introduction ............................................................................. 3
  Supported Firmware Versions ................................................. 3
  Enumerations ........................................................................... 4
  Structures ............................................................................... 5
  Functions ............................................................................... 5
  Classes .................................................................................. 20
  Benchmarks ............................................................................. 22
  Examples ............................................................................... 25
  Release Notes .......................................................................... 33
RTAC LIBRARY

PacketEncoding

Introduction

The PacketEncoding library provides functions and classes to decode from and encode to common data representations.

Various functions translate bytes of data to and from classes that facilitate storing information in an easy-to-use manner.

Special Considerations

Copying classes from this library causes unwanted behavior. This means the following:

1. The assignment operator "=" must not be used on any class from this library; consider assigning pointers to the objects instead.

   ```
   // This is bad and in most cases will provide a compiler error such as:
   // "C0328: Assignment not allowed for type class_VectorObject"
   myVectorObject := otherVectorObject;
   
   // This is fine
   someVariable := myVectorObject.value;
   // As is this
   pt_myVectorObject := ADR(myVectorObject);
   ```

2. Classes from this library must never be VAR_INPUT or VAR_OUTPUT members in function blocks, functions, or methods. Place them in the VAR_IN_OUT section or use pointers instead.

Supported Firmware Versions

You can use this library on any device configured using acSELERATOR RTAC® SEL-5033 Software with firmware version R143 or higher.

Versions 3.5.0.4 and older can be used on RTAC firmware version R132 and higher.
Enumerations

Enumerations make code more readable by allowing a specific number to have a readable textual equivalent.

**enum_Asn1ClassType**

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIVERSAL</td>
<td>0</td>
<td>The type is native to ASN.1.</td>
</tr>
<tr>
<td>APPLICATION</td>
<td>1</td>
<td>The type is only valid for one specific application.</td>
</tr>
<tr>
<td>CONTEXT_SPECIFIC</td>
<td>2</td>
<td>The meaning of this type depends on the context.</td>
</tr>
<tr>
<td>SPECIAL_PRIVATE</td>
<td>3</td>
<td>This type is defined in private specifications.</td>
</tr>
</tbody>
</table>

**enum_Asn1UniversalClassTags**

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOC</td>
<td>00</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>01</td>
</tr>
<tr>
<td>INTEGER</td>
<td>02</td>
</tr>
<tr>
<td>BIT_STRING</td>
<td>03</td>
</tr>
<tr>
<td>OCTET_STRING</td>
<td>04</td>
</tr>
<tr>
<td>NULL</td>
<td>05</td>
</tr>
<tr>
<td>OBJECT_IDENTIFIER</td>
<td>06</td>
</tr>
<tr>
<td>OBJECT_DESCRIPTOR</td>
<td>07</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>08</td>
</tr>
<tr>
<td>REAL_FLOAT</td>
<td>09</td>
</tr>
<tr>
<td>ENUMERATED</td>
<td>10</td>
</tr>
<tr>
<td>EMBEDDED_PDV</td>
<td>11</td>
</tr>
<tr>
<td>UTF8_STRING</td>
<td>12</td>
</tr>
<tr>
<td>RELATIVE_OID</td>
<td>13</td>
</tr>
<tr>
<td>RESERVED_1</td>
<td>14</td>
</tr>
<tr>
<td>RESERVED_2</td>
<td>15</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>16</td>
</tr>
<tr>
<td>SET</td>
<td>17</td>
</tr>
<tr>
<td>NUMERIC_STRING</td>
<td>18</td>
</tr>
<tr>
<td>PRINTABLE_STRING</td>
<td>19</td>
</tr>
<tr>
<td>T61_STRING</td>
<td>20</td>
</tr>
<tr>
<td>VIDEOTEX_STRING</td>
<td>21</td>
</tr>
<tr>
<td>IA5_STRING</td>
<td>22</td>
</tr>
<tr>
<td>UTC_TIME</td>
<td>23</td>
</tr>
<tr>
<td>GENERALIZED_TIME</td>
<td>24</td>
</tr>
<tr>
<td>GRAPHIC_STRING</td>
<td>25</td>
</tr>
<tr>
<td>VISIBLE_STRING</td>
<td>26</td>
</tr>
<tr>
<td>GENERAL_STRING</td>
<td>27</td>
</tr>
<tr>
<td>UNIVERSAL_STRING</td>
<td>28</td>
</tr>
<tr>
<td>CHARACTER_STRING</td>
<td>29</td>
</tr>
</tbody>
</table>
Structures

Structures provide a means to group together several memory locations (variables), making them easier to manage.

struct_Asn1Index

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>enum_Asn1ClassType</td>
<td>The class as defined by the first two Identifier bits.</td>
</tr>
<tr>
<td>Constructed</td>
<td>BOOL</td>
<td>True for constructed entries, False for primitive entries.</td>
</tr>
<tr>
<td>TagNumber</td>
<td>enum_Asn1UniversalClassTags</td>
<td>The type of the entry.</td>
</tr>
<tr>
<td>BytesInValue</td>
<td>UDINT</td>
<td>The number of content bytes.</td>
</tr>
<tr>
<td>Index</td>
<td>UDINT</td>
<td>The starting location of the content bytes as a byte offset from the beginning of the parsed byte array.</td>
</tr>
</tbody>
</table>

Functions

This library provides the following functions.

fun_IndexAsn1Packet

Walk the provided byte array and populate a class_Asn1IndexVector with the size, starting index, and type of each first level entry in an ASN.1 packet as defined by the “Basic Encoding Rules” (BER), the superset of encoding algorithms explained in the ASN.1 standard.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_data</td>
<td>POINTER TO BYTE</td>
<td>The packet to parse.</td>
</tr>
<tr>
<td>numBytes</td>
<td>UDINT</td>
<td>The number of bytes in the provided packet.</td>
</tr>
</tbody>
</table>

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parsedData</td>
<td>class_Asn1IndexVector</td>
<td>The vector for storing the list of data types and indices.</td>
</tr>
</tbody>
</table>
Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if all data objects were indexed successfully.</td>
</tr>
</tbody>
</table>

Processing

The `fun_IndexAsn1Packet()` function does the following:

- Validates `pt_data` for readability.
- Clears all data from `parsedData`.
- Locates the beginning of each first-level data entry in the packet.
- Stores the class, tag number, and length in bytes of each entry found as well as whether the entry is primitive or constructed.
- Stores the index zero for any objects of length zero.
- Stores the index of the first content byte of the entry as a byte offset from `pt_data` for all other data types.

This is accomplished by traversing the entire byte array from the beginning to the end. Any failure in parsing data results in an error and the function stops attempting to parse the provided data. The algorithm in use takes the first byte and interprets it to find the type of data being encoded. If the function finds a tag type of 0b11111 the next bytes are interpreted as the type of the data. If more than 32 bits of data are used to encode the type, then this method will return an error. The function interprets the next bytes as the length of the data.

Three length definitions are allowed:

1. A value less than 0x80 is a direct reference to the length. The function tags the next byte as the index and skips to the end of the object as defined by its length.
2. A value of 0x80 indicates that the length is not predefined. This value is only allowed for constructed types. The function tags the next byte as the index and immediately terminates the object. The function parses subsequent objects for length and ignores the content until it finds one End-of-Content object for each previously recorded length 0x80. The length becomes the accumulation of all the sizes of the child objects, including their headers.
3. A value between 0x81 and 0x84 indicates that one to four subsequent bytes define the length of the object as an unsigned integer. The function stores those bytes as the length and places the index directly after them. It then skips to the end of the object as defined by its length and the next object begins.

Though values larger than 0x84 are legal, they define numbers of bytes larger than this library can index and result in an error.

This process repeats for each object found until the end of the provided data. If the final length sends the function beyond the end of the array or unclosed length 0x80 objects remain, the function returns an error.

```
0101FF02038765430904A73546FF048180<128 Octets>
Becomes
0101FF BOOLEAN length 1 index 2
0203876543 INTEGER length 3 index 5
0904A73546FF REAL_FLOAT length 4 index 10
048180<128 Octets> OCTET_STRING length 128 index 17
```
fun_DecodeAsn1_Boolean

Decode a Boolean encoded in ASN.1 following the Basic Encoding Rules.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>BYTE</td>
<td>The byte to parse. This should be the byte at the index returned by <code>fun_IndexAsn1Packet()</code></td>
</tr>
</tbody>
</table>

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>BOOL</td>
<td>The result of parsing the data.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful parsing of the Boolean.</td>
</tr>
</tbody>
</table>

Processing

The `fun_DecodeAsn1_Boolean()` function does the following:

1. Parses the provided byte into a Boolean.
2. Returns TRUE if the function encountered no errors during parsing.

This function recognizes any non-zero value as TRUE and only the value of zero as FALSE.

0b11001100 = TRUE
0b00000001 = TRUE
0b00000000 = FALSE

fun_DecodeAsn1_Integer

Decode an integer encoded in ASN.1 following the "Basic Encoding Rules."

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_data</td>
<td>POINTER TO BYTE</td>
<td>The byte array to parse. This should be the address of the Index returned by <code>fun_IndexAsn1Packet()</code></td>
</tr>
<tr>
<td>numBytes</td>
<td>UDINT</td>
<td>The number of bytes in the provided data. This should be the <code>BytesInValue</code> returned by <code>fun_IndexAsn1Packet()</code></td>
</tr>
</tbody>
</table>
### Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>DINT</td>
<td>The result of parsing the data.</td>
</tr>
</tbody>
</table>

### Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful parsing of the integer.</td>
</tr>
</tbody>
</table>

### Processing

The `fun_DecodeAsn1_Integer()` function does the following:

2. Parses the bytes provided into an integer.
3. Returns FALSE if rollover prevents the function from returning the exact value.
4. Returns TRUE if the function encounters no errors during parsing.

This function expects the provided bytes to be an integer represented in two’s complement notation using the least number of bytes possible. It only parses numbers represented by four or fewer bytes.

0x80 = -128  
0xFF80 results in an error because it uses more bytes than necessary.  
0x7F7F = 32639  
0x007F7F results in an error because it uses more bytes than necessary.

### fun_DecodeAsn1_Enumerated

Decode an enumeration encoded in ASN.1 following the “Basic Encoding Rules.”

### Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_data</td>
<td>POINTER TO BYTE</td>
<td>The byte array to parse. This should be the address of then Index returned by <code>fun_IndexAsn1Packet()</code>.</td>
</tr>
<tr>
<td>numBytes</td>
<td>UDINT</td>
<td>The number of bytes in the provided data. This should be the BytesInValue returned by <code>fun_-IndexAsn1Packet()</code>.</td>
</tr>
</tbody>
</table>

### Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>DINT</td>
<td>The result of parsing the data.</td>
</tr>
</tbody>
</table>
Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful parsing of the enumeration.</td>
</tr>
</tbody>
</table>

Processing

The `fun_DecodeAsn1_Enumerated()` function does the following:

2. Parses the bytes provided into an integer.
3. Returns FALSE if rollover prevents returning the exact value.
4. Returns TRUE if the function encountered no errors during parsing.

This function expects the provided bytes to be an integer represented in two's complement notation using the least number of bytes possible. It only parses numbers represented by four or fewer bytes.

<table>
<thead>
<tr>
<th>x80 = -128</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFF80 results in an error because it uses more bytes than necessary.</td>
</tr>
<tr>
<td>0x7FF7F = 32639</td>
</tr>
<tr>
<td>0x007FF7F results in an error because it uses more bytes than necessary.</td>
</tr>
</tbody>
</table>

`fun_DecodeAsn1_Object_Identifier`

Decode an Object Identifier (OID) encoded in ASN.1 following the “Basic Encoding Rules” to a list of UDINTs.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_data</td>
<td>POINTER TO BYTE</td>
<td>The byte array to parse. This should be the address of the index returned by <code>fun_IndexAsn1Packet()</code>.</td>
</tr>
<tr>
<td>numBytes</td>
<td>UDINT</td>
<td>The number of bytes in the provided data. This should be the <code>BytesInValue</code> returned by <code>fun_IndexAsn1Packet()</code>.</td>
</tr>
</tbody>
</table>

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>class_DwordVector</td>
<td>The vector for storing the list of OID entries.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful parsing of the OID.</td>
</tr>
</tbody>
</table>
Processing

The `fun_DecodeAsn1_Object_IDENTIFIER()` function does the following:

2. Clears all data from `value`.
3. Parses the bytes provided into an OID.
4. Returns FALSE if rollover prevents the function from returning exact values.
5. Returns TRUE if the function encountered no errors during parsing.

This function expects the provided bytes to appear as a sequence of unsigned integers represented in the fewest bytes possible. Any byte with the value 1 as its most significant bit indicates that the next byte is part of the same integer value. A value of one in the most significant bit of the final byte of the referenced data indicates the OID would extend beyond `numBytes` and the function returns FALSE. If any unsigned integer begins with 0x80, the number is represented by more bytes than required and the function returns FALSE. The function also returns FALSE if any unsigned integer requires more than 32 bits to contain its value.

After finding the first unsigned integer, the function parses it into two distinct values. For this example, consider `val` to be the first unsigned integer found. If `val` is between 0 and 39, the OID begins with 0.val. If `val` is between 40 and 79, the OID begins with 1.(val-40). Finally, if `val` is greater than 79 the OID begins with 2.(val-80).

\[ \text{0x2B}\text{06}\text{01}\text{04}\text{01}\text{8F}\text{84}\text{F0}\text{19}\text{52}\text{AD}\text{04}\text{01}00 \]

Broken into parts

\[
\begin{align*}
0x2B & \rightarrow 40 \& 3 \\
0x06 & \rightarrow 0b111_1000_100_1111 \rightarrow 31823 \\
0x01 & \rightarrow 0b1_0101_010_1010 \rightarrow 2730 \\
0x8F\text{84}\text{F0} & \rightarrow 0b1\_111\_1000\_100\_1111 \rightarrow 31823 \\
0x95\text{2A} & \rightarrow 0b1\_0101\_010\_1010 \rightarrow 2730 \\
\text{1.3.6.1.4.1.31823.1.2730.13.4.1.0} & \\
\end{align*}
\]

### fun_DecodeAsn1_Real

Decode a floating point number encoded as ASN.1 in base 2, 8, or 16 according to the “Basic Encoding Rules.”

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| pt_data| POINTER TO BYTE| The byte array to parse. This should be the address of the index returned by `fun_IndexAsn1Packet()`.
| numBytes| UDINT         | The number of bytes in the provided data. This should be the `BytesInValue` returned by `fun_IndexAsn1Packet()`.

---

PacketEncoding Instruction Manual Date Code 20180924
Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>REAL</td>
<td>The decoded floating point number.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful parsing of the number.</td>
</tr>
</tbody>
</table>

Processing

The `fun_DecodeAsn1_Real()` function does the following:

2. Validates the format and base found in `pt_data`.
3. Parses the bytes provided into a real.
4. Returns TRUE if the function encounters no errors during parsing.

The ASN.1 standard allows reals to be encoded through the use of both binary values and character-based encodings. This function only decodes reals that are encoded as binary values in base 2, 8, or 16.

The decoding of a real happens in eight steps:

1. This function checks for any special values (see Special Bit Patterns for Reals for details).
2. If `numBytes` is two, this function returns FALSE because there are no valid real encodings of length two.
3. This function checks the first byte to ensure it can be parsed. Bit 8 must be one. Bit 7 stores the sign of the value. Bits 6–5 represent the base of the number (B): 0b00 is base 2, 0b01 is base 8, 0b10 is base 16, and 0b11 is invalid. Bits 4–3 are interpreted as an unsigned number defining a power of two by which to shift the result (F). Bits 2–1 help delineate the length of the exponent in bytes. A value of 0b00 means one byte of exponent, 0b01 two bytes, 0b10 three bytes, and 0b11 means the next byte defines the length as a value from 0–255.
4. Because ASN.1 states the exponent must be represented in the fewest bytes possible, this function rounds any value with an exponent of three or more bytes to positive/negative zero or infinity because it cannot represent numbers that large or small.
5. The exponent is parsed as a two’s complement number (E) and saved.
6. The function interprets as many as the first four of the remaining bytes as an unsigned integer.
7. Each byte greater than four results in the addition of eight to F, truncating the mantissa (M) to a size this function can handle.
8. Finally, the function calculates the result as \((M \cdot 2^F \cdot B^E)\), then adds the appropriate sign.
fun_SerializeAsn1_Boolean

Place the provided Boolean value as the next entry in an ASN.1 BER message.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>BOOL</td>
<td>The value to use as the payload of this Boolean.</td>
</tr>
</tbody>
</table>

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>class.ByteVector</td>
<td>The message to which value is appended.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful appending of the Boolean.</td>
</tr>
</tbody>
</table>

Processing

The fun_SerializeAsn1_Boolean() function does the following:

1. Serializes value and appends it as the next entry in message.
2. Returns TRUE if value was added to message.

This function appends 0x010100 for false and 0x010101 for true.

fun_SerializeAsn1_Integer

Place the integer value provided as the next entry in an ASN.1 BER message.
## Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>DINT</td>
<td>The value to use as the payload of this integer.</td>
</tr>
</tbody>
</table>

## Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>class_BYTEVECTOR</td>
<td>The message to which value is appended.</td>
</tr>
</tbody>
</table>

## Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful appending of the integer.</td>
</tr>
</tbody>
</table>

## Processing

The `fun_SerializeAsn1_Integer()` function does the following:

1. Serializes value and appends it as the next entry in message.
2. Returns TRUE if value was added to message.

This function determines the fewest number of bytes necessary to encode the provided integer by checking the most significant byte and the next bit for all ones or all zeros and dropping the byte from the number. This process can be repeated as many as three times. After this process completes, it appends the following to the message: 0x02, the number of significant bytes, and the bytes themselves.

<table>
<thead>
<tr>
<th>Value</th>
<th>Byte String</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0x0201FF</td>
</tr>
<tr>
<td>1</td>
<td>0x0200101</td>
</tr>
<tr>
<td>134217728</td>
<td>0x0200F000000</td>
</tr>
<tr>
<td>-134217728</td>
<td>0x0200F8000000</td>
</tr>
</tbody>
</table>

## fun_SerializeAsn1_Enumerated

Place the integer value provided as the next entry in an ASN.1 BER message.

## Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>DINT</td>
<td>The value to use as the payload of this enumeration.</td>
</tr>
</tbody>
</table>


## Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>class_ByteVector</td>
<td>The message to which value is appended.</td>
</tr>
</tbody>
</table>

## Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful appending of the enumeration.</td>
</tr>
</tbody>
</table>

## Processing

The `fun_SerializeAsn1_Enumerated()` function does the following:

1. Serializes value and appends it as the next entry in message.
2. Returns TRUE if value was added to message.

This function determines the fewest number of bytes necessary to encode the provided integer by checking the most significant byte and the next bit for all ones or all zeros and dropping the byte from the number. This process can be repeated as many as three times. After this process completes, it appends the following to the message: 0x02, the number of significant bytes, and the bytes themselves.

<table>
<thead>
<tr>
<th>-1</th>
<th>0x0201FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x020101</td>
</tr>
<tr>
<td>134217728</td>
<td>0x02040F000000</td>
</tr>
<tr>
<td>-134217728</td>
<td>0x0204F8000000</td>
</tr>
</tbody>
</table>

## fun_SerializeAsn1_Real

Place the real provided as the next entry in an ASN.1 BER message.

## Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>REAL</td>
<td>The value to use as the payload of this real.</td>
</tr>
</tbody>
</table>

## Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>class_ByteVector</td>
<td>The message to which value is appended.</td>
</tr>
</tbody>
</table>
### Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful appending of the real.</td>
</tr>
</tbody>
</table>

### Processing

The `fun_SerializeAsn1_Real()` function does the following:

1. Serializes `value` as a base 2 floating point number and appends it as the next entry in `message`.
2. Returns true if `value` was added to `message`.

This function serializes every real as a base 2 binary encoded real. First, it checks for any special value encodings as listed in *Special Bit Patterns for Reals*. If this real is not a special case, this method breaks the real into its constituent parts. This process consists of four steps:

1. The sign is pulled from Bit 32, the exponent from Bits 31 to 24, and the mantissa from Bits 23 to 1.
2. If the exponent is zero, the mantissa is saved as is. Otherwise the function prepends a one to the mantissa as the 24th bit.
3. If the exponent is zero, a value of one is added to it. Then the function subtracts 127 subtracted from the exponent to turn it into a two’s complement, signed number; the function also subtracts 23 from it to remove the decimal point from the mantissa.
4. The function generates a descriptive byte where Bit 8 is one, Bit 7 is one for negative and zero for positive, Bits 6–2 are zero, and Bit 1 is one for an exponent requiring a two byte representation and zero for an exponent requiring one byte.

This function then appends 0x09; a length of 0x00, 0x01, 0x05 or 0x06; the descriptive byte; the exponent; and the mantissa to `message`.

<table>
<thead>
<tr>
<th>Value</th>
<th>IEEE Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.25</td>
<td>0x090580EA900000</td>
</tr>
<tr>
<td>-2.25</td>
<td>0x0905C0EA900000</td>
</tr>
<tr>
<td>1.25E-38</td>
<td>0x090681FF6B81CEA</td>
</tr>
<tr>
<td>1.25E-41</td>
<td>0x090681FF6B0022D8</td>
</tr>
</tbody>
</table>

### `fun_SerializeAsn1_Object_Identifier`

Place the OID provided as the next entry in an ASN.1 BER message.

#### Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>class_DwordVector</td>
<td>The OID to append to <code>message</code>.</td>
</tr>
<tr>
<td>message</td>
<td>class_ByteVector</td>
<td>The message to which <code>value</code> is appended.</td>
</tr>
</tbody>
</table>
### Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful appending of the OID.</td>
</tr>
</tbody>
</table>

### Processing

The `fun_SerializeAsn1_Object_Identifier()` function does the following:

1. Serializes all dwords in `value` to construct the next entry in `message`.
2. Returns FALSE if `value` cannot be serialized.
3. Returns TRUE if `value` was added to `message`.

This function starts by adding the first two OID sub entries together \((40 \cdot OID1) + OID2\). Using that result as the first sub entry, it builds a list where each OID sub entry is reduced to the minimum number of seven-bit segments needed to represent it as an unsigned value. The function prepends a one to each seven-bit segment except the least significant seven-bit segment of every sub entry, which the function prepends with a zero. Once all sub entries have been encoded, the function appends the following to `message`: 0x06, the number of bytes representing the sub entries encoded in the same manner as the sub entries themselves, and the sub entry list.

```
1.3.6.1.4.1.31823.1.2730.13.4.4.213268340
Insert Extra formatting
OID Enumeration     => 0x06
1.3                   => 40 & 3       => 0x2B
.6.1.4.1              => 0x06010401
.31823                => 0x7C4F       => 0x81F84F
.1                    => 0x01
.2730                 => 0x0AAA      => 0x952A
.13.4.4               => 0x00D0404
.213268340            => 0xCB63774    => 0xE5D8EE74
length                => 18         => 0x12
Construct as Enumeration, Length, SubEntry list
0x06122B0601040181F84F01952A00D0404E5D8EE74
```

### `fun_SerializeAsn1_Bit_String`

Place the bit string provided as the next entry in an ASN.1 BER message.

### Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_data</td>
<td>POINTER TO BYTE</td>
<td>The bit string to insert.</td>
</tr>
<tr>
<td>numBytes</td>
<td>UDINT</td>
<td>The number of bytes in the string.</td>
</tr>
<tr>
<td>ignoreBits</td>
<td>USINT</td>
<td>The number of bits of invalid data terminating the string. (Range: 0–7)</td>
</tr>
</tbody>
</table>
PacketEncoding

Functions

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>class_ByteVector</td>
<td>The message to which the bit string is appended.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful appending of the bit string</td>
</tr>
</tbody>
</table>

Processing

The fun_SerializeAsn1_Bit_String() function does the following:

1. Validates pt_data for read access.
2. Limits ignoreBits to a maximum of seven.
3. Appends an entry containing all bytes prescribed to message, masking the last ignoreBits bits of the last byte by replacing them with zero.
4. Returns true if the entire bit string was added to message.

This function appends 0x03, numBytes plus one for the ignoreBits information as the length, ignoreBits, and the data found at pt_data to message, zeroing the final ignoreBits bits.

fun_BeginAsn1Constructed_Bit_String

Place the codes necessary to begin a constructed bit string in an ASN.1 BER message.

For each call to this method the user must call fun_AppendAsn1_Eoc() before the message can be considered complete. Only bit strings should be appended to message until the call to fun_AppendAsn1_Eoc().

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>class_ByteVector</td>
<td>The message to which the entry is appended.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful appending of the entry.</td>
</tr>
</tbody>
</table>

Processing

The fun_BeginAsn1Constructed_Bit_String() function:

1. Appends the beginning of a constructed bit string to message, (0x2380).
2. Returns TRUE if the entry was added to message.
fun_AppendAsn1_Eoc

Place the codes necessary to end a variable length entry in an ASN.1 BER message.
To ensure proper packet construction, the user must call this function once for each variable
length entry begun.

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>class_ByteVector</td>
<td>The message to which the entry is appended.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE for successful appending of the EOC.</td>
</tr>
</tbody>
</table>

Processing

The `fun_AppendAsn1_Eoc()` function does the following:
1. Appends the End-of-Content entry to `message`, 0x0000).
2. Returns TRUE if the EOC was added to `message`.

fun_EncodeBase64_MIME

Encodes a byte vector into base64-MIME format. See base64 and MIME descriptions in
RFC 2045 for complete definition of these encodings and their usage. A common example
is encoding the bytes of a file to be sent as email attachments.

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>class_ByteVector</td>
<td>The raw byte data to encode.</td>
</tr>
<tr>
<td>encoded</td>
<td>class_ByteVector</td>
<td>The encoded output of <code>source</code>.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if data was successfully encoded; returns FALSE only if <code>source</code> was empty.</td>
</tr>
</tbody>
</table>
Processing

The `fun_EncodeBase64_MIME()` function does the following:

1. Encodes the raw-bytes of `source`, writing the base64-MIME encoded output to `encoded`. Note that `source` will not be modified as a result of calling this function.
2. Returns TRUE if input was encoded successfully; returns FALSE only if `source` was empty.

This function encodes a raw byte vector in base64-MIME. The output `encoded` will be approximately 133% the size of `source`.

```plaintext
source := Drink plenty of Ovaltine
encoded := RHJpbmsgcGx1bnR5IG9mIE92YWx0aW5l
```

`fun_DecodeBase64_MIME`

Decodes a byte vector encoded in base64-MIME format. See base64 and MIME descriptions in RFC 2045 for complete definition of these encodings and their usage.

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>class_ByteVector</td>
<td>The base64-MIME encoded data to decode.</td>
</tr>
<tr>
<td>decoded</td>
<td>class_ByteVector</td>
<td>The decoded output of source.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if data was successfully decoded without any corruption detected.</td>
</tr>
<tr>
<td></td>
<td>Returns FALSE if <code>source</code> contains only invalid characters or if the number</td>
</tr>
<tr>
<td></td>
<td>of valid characters in <code>source</code> is not a multiple of four.</td>
</tr>
</tbody>
</table>

Processing

The `fun_DecodeBase64_MIME()` function does the following:

1. Decodes the base64-MIME encoded input of `source`, placing output in `decoded`. Invalid, non-base64 characters in `source` are ignored. Note that `source` will not be modified as a result of calling this function.
2. Processes valid base64 characters in groups of four. It fails only if terminal characters are incorrectly placed or if the number of valid characters is not a multiple of four.
3. Stops processing after the first group of four characters containing a terminal character.
4. If `source` is empty, then the function will return TRUE.

This function decodes a base64-MIME encoded string. The size of `decoded` will be approximately 75% of `source`. 
Classes

Classes are a particular implementation of a Function Block (FB). They provide Methods and Properties, which a normal FB does not provide.

class_AsniIndexVector (Class)

This class provides a DynamicVector structured specifically to store indexing information about a byte array encoded in ASN.1.

Implemented Interfaces

An interface defines a required set of functionality as methods and properties. As an implementer of any interface all methods and properties declared in that interface must exist as members of this class. This allows multiple generally unrelated classes to be used interchangeably for a specific feature set.

➤ I_Vector

GetAt (Method)

Provides a copy of the element at the specified index.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>UDINT</td>
<td>The index of the desired element in the vector.</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td>struct_AsniIndex</td>
<td>The element at the specified index. If the return value is FALSE, this value is undefined.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if index is valid and the element is copied. FALSE if index is invalid or an error occurs.</td>
</tr>
</tbody>
</table>
SetAt (Method)

This method provides write access to any element within the vector.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>UDINT</td>
<td>The index at which to set the value of an element.</td>
</tr>
<tr>
<td>value</td>
<td>struct_Asn1Index</td>
<td>The new element value.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the element is successfully modified. If index is invalid, the vector is not modified and the method returns FALSE.</td>
</tr>
</tbody>
</table>

Pop (Method)

This method provides a copy of the last item in the vector and removes that element from the vector.

Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td>struct_Asn1Index</td>
<td>A copy of the former last element in the vector. If the return value is FALSE, this value is undefined.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if element is successfully copied and removed from the vector. FALSE if the size is zero or an error occurs.</td>
</tr>
</tbody>
</table>

Push (Method)

This method appends a copy of the provided element to the end of the vector.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td>struct_Asn1Index</td>
<td>The element to copy to the end of the vector.</td>
</tr>
</tbody>
</table>
Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the element is successfully added to the vector. FALSE if an error occurs.</td>
</tr>
</tbody>
</table>

Benchmarks

Benchmark Platforms

The benchmarking tests recorded for this library are performed on the following platforms.

- SEL-3505
  - R134 firmware
- SEL-3530
  - R134 firmware
- SEL-3555
  - Dual-core Intel i7-3555LE processor
  - 4 GB ECC RAM
  - R134-V1 firmware

Benchmark Test Descriptions

fun_IndexAsn1Packet

The posted time is the average execution time of 100 consecutive calls. The byte string parsed is loaded with two Boolean values and three integer values.

fun_DecodeAsn1_Boolean

The posted time is the average execution time of 100 consecutive calls.

fun_DecodeAsn1_Integer

The posted time is the average execution time of 100 consecutive calls.

fun_DecodeAsn1_ENUMerated

The posted time is the average execution time of 100 consecutive calls.
fun_DecodeAsn1_Object_Identifier
The posted time is the average execution time of 100 consecutive calls. The encoded object identifier that is decoded has a value of (1, 17, 19).

fun_DecodeAsn1_Real
The posted time is the average execution time of 100 consecutive calls.

fun_SerializeAsn1_Boolean
The posted time is the average execution time of 100 consecutive calls.

fun_SerializeAsn1_Integer
The posted time is the average execution time of 100 consecutive calls.

fun_SerializeAsn1_Enumerated
The posted time is the average execution time of 100 consecutive calls.

fun_SerializeAsn1_Real
The posted time is the average execution time of 100 consecutive calls.

fun_SerializeAsn1_Object_Identifier
The posted time is the average execution time of 100 consecutive calls. The object identifier encoded has a value of (1, 17, 19).

fun_SerializeAsn1_Bit_String
The posted time is the average execution time of 100 consecutive calls. The bit string encoded has a ASCII value of “Hello World”.

fun_BeginAsn1Constructed_Bit_String
The posted time is the average execution time of 100 consecutive calls.

fun_AppendAsn1_Eoc
The posted time is the average execution time of 100 consecutive calls.
fun_EncodeBase64_MIME - No Memory Allocation

The posted time is the average execution time of 100 consecutive calls when encoding a sequence of 100 random bytes. For this benchmark, the encoded vector is sized such that no memory allocation is required during the benchmark run.

fun_EncodeBase64_MIME - Internal Memory Allocation

The posted time is the average execution time of 100 consecutive calls when encoding a sequence of 100 random bytes. For this benchmark, the encoded vector is empty with no memory allocated, thus requiring memory allocations during the benchmark run.

fun_DecodeBase64_MIME - No Memory Allocation

The posted time is the average execution time of 100 consecutive calls when decoding a sequence of 100 randomly encoded bytes. Note that because the output is 100 bytes, the input vector is more than 100 bytes. For this benchmark, the decoded vector is sized such that no memory allocation is required during the benchmark run.

fun_DecodeBase64_MIME - Internal Memory Allocation.

The posted time is the average execution time of 100 consecutive calls when decoding a sequence of 100 randomly encoded bytes. Note that because the output is 100 bytes, the input vector is more than 100 bytes. For this benchmark, the encoded vector is empty with no memory allocated, thus requiring memory allocations during the benchmark run.

Benchmark Results

<table>
<thead>
<tr>
<th>Operation Tested</th>
<th>Platform (time in µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SEL-3505</td>
</tr>
<tr>
<td>fun_IndexAsn1Packet</td>
<td>30</td>
</tr>
<tr>
<td>fun_DecodeAsn1_Boolean</td>
<td>1</td>
</tr>
<tr>
<td>fun_DecodeAsn1_Integer</td>
<td>3</td>
</tr>
<tr>
<td>fun_DecodeAsn1_Enumerated</td>
<td>2</td>
</tr>
<tr>
<td>fun_DecodeAsn1_Object_Identifier</td>
<td>19</td>
</tr>
<tr>
<td>fun_DecodeAsn1_Real</td>
<td>10</td>
</tr>
<tr>
<td>fun_SerializeAsn1_Boolean</td>
<td>4</td>
</tr>
<tr>
<td>fun_SerializeAsn1_Integer</td>
<td>8</td>
</tr>
<tr>
<td>fun_SerializeAsn1_Enumerated</td>
<td>8</td>
</tr>
<tr>
<td>fun_SerializeAsn1_Real</td>
<td>6</td>
</tr>
<tr>
<td>fun_SerializeAsn1_Object_Identifier</td>
<td>159</td>
</tr>
<tr>
<td>fun_SerializeAsn1_Bit_String</td>
<td>18</td>
</tr>
<tr>
<td>fun_BeginAsn1Constructed_Bit_String</td>
<td>8</td>
</tr>
<tr>
<td>fun_AppendAsn1_Eoc</td>
<td>6</td>
</tr>
<tr>
<td>fun_EncodeBase64_MIME - No Allocation</td>
<td>432</td>
</tr>
<tr>
<td>fun_EncodeBase64_MIME - Allocation</td>
<td>637</td>
</tr>
</tbody>
</table>
### Examples

*These examples demonstrate the capabilities of this library. Do not mistake them as suggestions or recommendations from SEL.*

Implement the best practices of your organization when using these libraries. As the user of this library, you are responsible for ensuring correct implementation and verifying that the project using these libraries performs as expected.

#### Decoding an ASN.1 Packet for All Integer Values

**Objective**

A user has a system that sends packets of data containing a mixture of integer values and octet string descriptions. She needs to use the integers to make decisions but has no need of the strings on this RTAC. This solution parses the packet and collects the four integer values it contains.

**Assumptions**

This example shows the parsing of a static byte array. To truly use this functionality, the user would need to populate that array after collecting the data from the network.

**Solution**

The user can create the program found in *Code Snippet 1* to parse the byte array.

<table>
<thead>
<tr>
<th>Operation Tested</th>
<th>Platform (time in µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SEL-3505</td>
</tr>
<tr>
<td>fun_DecodeBase64_MIME - No Allocation</td>
<td>430</td>
</tr>
<tr>
<td>fun_DecodeBase64_MIME - Allocation</td>
<td>699</td>
</tr>
</tbody>
</table>
Code Snippet 1  prg_ParseBytesForIntegers

PROGRAM prg_ParseBytesForIntegers
VAR
   PacketBytes : ARRAY [0 .. 99] OF BYTE :=
   [
      (*The string Value1*)
      16#04, 16#06, 16#56, 16#61, 16#6C, 16#75, 16#65, 16#31,
      (*The integer 10_000*)
      16#02, 16#02, 16#27, 16#10,
      (*The string Value2*)
      16#04, 16#06, 16#61, 16#6C, 16#75, 16#65, 16#32,
      (*The integer -50_000*)
      16#02, 16#03, 16#FF, 16#3C, 16#B0,
      (*The string Value3*)
      16#04, 16#06, 16#56, 16#61, 16#6C, 16#75, 16#65, 16#33,
      (*The integer 15*)
      16#02, 16#01, 16#0F,
      (*The string Value4*)
      16#04, 16#06, 16#56, 16#61, 16#6C, 16#75, 16#65, 16#34,
      (*The integer 1_500_000_000*)
      16#02, 16#04, 16#59, 16#68, 16#2F, 16#00,
      (*This is the end of the meaningful data. The array here is bigger than
       required as a reminder that this may need to be populated with
different values that take more or less space.*)
      50(0)];
   // The number of bytes of valid data. This will come from the network
   // socket.
   ValidByteCount : UDINT := 50;

   // Containers for parsed data.
   IndexList : class_Asn1IndexVector;
   IndexObject : struct_Asn1Index;

   // Iterator counts.
   ListPosition : UDINT;
   ObjectCount : UDINT;

   //The result array.
   IntArray : ARRAY [0 .. 3] OF DINT;

   // Flags for any errors that might be encountered.
   Parsed : BOOL;
   CorrectCount : BOOL;
   ValidInts : ARRAY [0 .. 3] OF BOOL;
END_VAR
Decoding an OID Found Three Layers Deep in an ASN.1 Packet

Objective

A user receives a package with an OID nested three layers deep inside. He needs to decode that OID before making a work decision.

Assumptions

This example shows the parsing of a static byte array. To truly use this functionality the user would need to populate that array after collecting the data from the network.
Solution

The user can create the program found in *Code Snippet 2* to parse the byte array.

**Code Snippet 2  prg_ParseThreeTiers**

```plaintext
PRG prg_ParseThreeTiers
VAR
  PacketBytes : ARRAY [0..99] OF BYTE :=
    [ (*Constructed sequence*)
      16#30, 16#13,
      (*Constructed sequence*)
      16#30, 16#11,
      (*OID 1.3.6.1.4.1.31823.1.2730.13.4.1.0*)
      16#06, 16#0F, 16#2B, 16#06, 16#01, 16#04, 16#01, 16#81, 16#F8,
      16#4F, 16#01, 16#95, 16#2A, 16#0D, 16#04, 16#01, 16#00,
      (*This is the end of the meaningful data
       The array here is bigger than required as a reminder that
       This may need to be populated with different values that take
       more or less space. *)
      79(0)];
  // The number of bytes of valid data. This will come from the network
  // socket.
  ValidByteCount : UDINT := 21;

  // Containers for each tier.
  Tier1Objects : class_Asn1IndexVector;
  Tier2Objects : class_Asn1IndexVector;
  Tier3Objects : class_Asn1IndexVector;
  IndexObjectT1 : struct_Asn1Index;
  IndexObjectT2 : struct_Asn1Index;
  IndexObjectT3 : struct_Asn1Index;

  // Flags to allow for separation of logic
  ParsedL1 : BOOL;
  ParsedL2 : BOOL;
  ParsedL3 : BOOL;
  ValidDid : BOOL;

  // Container for the OID
  Oid : PacketEncodings.class_DwordVector;
END_VAR
```
Code Snippet 2  prg_ParseThreeTiers (Continued)

// Reset from last scan
ParsedL2 := FALSE;
ParsedL3 := FALSE;
ValidOid := FALSE;

// Parse the first tier.
ParsedL1 := fun_IndexAsn1Packet(ADR(PacketBytes), ValidByteCount, Tier1Objects);
IF ParsedL1 THEN
    IF Tier1Objects.GetAt(0, element => IndexObjectT1) THEN
        IF IndexObjectT1.Constructed THEN
            ParsedL2 := fun_IndexAsn1Packet(
                ADR(PacketBytes[IndexObjectT1.Index]),
                IndexObjectT1.BytesInValue, Tier2Objects);
                END_IF
        ENDF
    ENDF
ENDIF

IF ParsedL2 THEN
    IF Tier2Objects.GetAt(0, element => IndexObjectT2) THEN
        IF IndexObjectT2.Constructed THEN
            ParsedL3 := fun_IndexAsn1Packet(
                ADR(PacketBytes[IndexObjectT1.Index + IndexObjectT2.Index]),
                IndexObjectT2.BytesInValue, Tier3Objects);
                END_IF
        ENDF
    ENDF
ENDIF

IF ParsedL3 THEN
    IF Tier3Objects.GetAt(0, element => IndexObjectT3) THEN
        IF IndexObjectT3.Class = UNIVERSAL AND
            IndexObjectT3.Asn1Class = OBJECT_IDENTIFIER THEN
            ValidOid := fun_DecodeAsn1_Object_Identifier(
                ADR(PacketBytes[IndexObjectT1.Index + IndexObjectT2.Index + IndexObjectT3.Index]),
                IndexObjectT3.BytesInValue, Oid);
            END_IF
        ENDF
    ENDF
ENDIF

IF ValidOid THEN
    ; // Do any necessary work based on the OID here.
END_IF
Encoding Data as an ASN.1 Packet

Objective

A user needs to package some integer and real data points. She also needs to intersperse the
data with Boolean flags based on the validity of the data.

Once the data are packaged the user needs to send the information to Port 1000 of a listening
server.

Assumptions

The RTAC also has access to the SELEthernetControllers library. The listening server must
know the format of the incoming data.

Solution

The user can create the program found in Code Snippet 3 to build the data package.

**Code Snippet 3  prg_EncodeOutboundData**

```plaintext
PROGRAM prg_EncodeOutboundData
VAR
   // The storage for the outbound packet.
   Packet : PacketEncodings.class_ByteVector;

   // The integers to encode.
   MyInts : ARRAY [1 .. 3] OF DINT := [-70, 45, 9000];
   MyIntsValid : ARRAY [1 .. 3] OF BOOL := [FALSE, TRUE, TRUE];

   // The reals to encode.
   MyReals : ARRAY [1 .. 3] OF REAL := [1045.99, 45.2, 7];
   MyRealsValid : ARRAY [1 .. 3] OF BOOL := [TRUE, FALSE, TRUE];

   // Infrastructure required to place the packet on the wire.
   MySocket : class_TcpClient;
   LocalIP : SELEthernetController.INADDR := (ulAddr := 0);
   DestinationIP : SELEthernetController.INADDR;
   SocketInitialized : BOOL;

   // Counting variable.
   i : UDINT;
END_VAR
```
// Make sure the socket is configured correctly.
IF NOT SocketInitialized THEN
    MySocket.bootstrap_SetLocalIP(1000, localIP);
    fun_StringToInaddr('10.10.10.10', ipAddress => DestinationIP);
    MySocket.SetIP(destinationIP, 1000);
    MySocket.Open();
    SocketInitialized := TRUE;
END_IF

// Build the packet.
Packet.Recycle();
FOR i := 1 TO 3 DO
    fun_SerializeAsn1_Integer(myInts[i], Packet);
    fun_SerializeAsn1_Boolean(myIntsValid[i], Packet);
END_FOR
FOR i := 1 TO 3 DO
    fun_SerializeAsn1_Real(myReals[i], Packet);
    fun_SerializeAsn1_Boolean(myRealsValid[i], Packet);
END_FOR

// Send the data.
MySocket.SendData(Packet.pt_Data, UDINT_TO_DINT(Packet.Size));

Encoding Raw Binary Data in Base64-MIME

Objective

A user needs to encode and store some binary data from the file system in base64-MIME encoding. For illustrative purposes, the user saves their data to a file.

Assumptions

The RTAC has access to the DynamicVectors and FileIo libraries.

Solution

The user can create the program found in Code Snippet 4 to encode their data in base64-MIME and store the encoded data to the local file system.
Code Snippet 4  prg_Base64_Example

PROGRAM prg_Base64_Example
VAR_INPUT
  alarm : BOOL;
END_VAR

VAR
  //State flags for file io operations
  complete : BOOL := FALSE;
  firstRead : BOOL := TRUE;
  writeFile : BOOL := FALSE;
  error : BOOL;
  errorString : STRING(255);

  //Location of the binary data the user wishes to encode
  dataFile : STRING(255) := 'binaryData.data';
  //Temporary location for the raw binary data
  rawData : class_ByteVector;
  //Temporary location for the encoded data
  encodedData : class_ByteVector;

  //File io objects
  fileReader : class_FilerReader();
  fileWriter : class_Filewriter('encodedData.txt');
END_VAR

IF alarm THEN
  IF NOT complete THEN
    IF firstRead THEN
      fileReader.ReadFile(dataFile);
      firstRead := FALSE;
    ELSIF fileReader(BytesInBuffer > 0 THEN
      rawData.Recycle();
      fileReader.AppendToVector(0, rawData);
    END_IF
    //encode the raw data in base64-MIME
    PacketEncodings.fun_EncodeBase64_MIME(source := rawData, encoded := encodedData);
    writeFile := TRUE;
  END_IF

  IF writeFile THEN
    fileWriter.AppendVector(encodedData);
    writeFile := FALSE;
    complete := TRUE;
  END_IF

  //log any errors
  IF fileReader.Error THEN
    error := TRUE;
    errorString := fileReader.ErrorDesc;
  END_IF
  IF fileWriter.Error THEN
    error := TRUE;
    errorString := fileWriter.ErrorDesc;
  END_IF
END_IF

//called each scan to complete the read/write ops
fileReader.Run();
fileWriter.Run();
## Release Notes

<table>
<thead>
<tr>
<th>Version</th>
<th>Summary of Revisions</th>
<th>Date Code</th>
</tr>
</thead>
</table>
| 3.5.1.0 | ➤ Allows new versions of ACSELERATOR RTAC to compile projects for previous firmware versions without SEL IEC types “Cannot convert” messages.  
➤ Replaced the deprecated “POINTER_TO_ANY” type with “POINTER_TO_BYTE”.  
➤ Must be used with R143 firmware or later.                                      | 20180921  |
| 3.5.0.4 | ➤ Added base64-MIME encoding and decoding functionality.                              | 20150722  |
| 3.5.0.3 | ➤ Initial release.                                                                    | 20141010  |