Chapter

8

Tutorial: Using ASN.1 Data Types

This tutorial describes how to use ASN.1 types and values in the SDL suite. You will learn how to import and use ASN.1 modules in your SDL diagrams, how to generate code and how to encode and decode your ASN.1 types using BER or PER encoding.

The tutorial contains all steps from creating ASN.1 data types to the implementation of the ASN.1 data types in your source code.

To illustrate the functionality and the work flow, small examples are presented throughout the tutorial. The SNMP protocol is used as a base to illustrate how ASN.1 could be applied on a typical SNMP stack.

In order for you to fully take advantage of this tutorial, you should be familiar with the SDL suite and the basics of ASN.1.

Additional information regarding ASN.1 types and its usage together with the SDL suite can be obtained in:

- **chapter 2, *Data Types, in the SDL Suite Methodology Guidelines***
- **chapter 14, *The ASN.1 Utilities, in the User's Manual***
- **chapter 58, *Building an Application, in the User's Manual***
- **chapter 59, *ASN.1 Encoding and De-coding in the SDL Suite, in the User's Manual***

Introduction

The Abstract Syntax Notation One (ASN.1) is a notation language that is used for describing structured information that is intended to be transferred across some type of interface or communication medium. It is especially used for defining communication protocols.

As ASN.1 is widely popular, the SDL suite allows you to translate ASN.1 data types to SDL and to encode/decode ASN.1 data types.

By using ASN.1 data types in the implementation of your application, you will optimize your development process. The following list displays some of the advantages of ASN.1:

- ASN.1 is a standardized, vendor-, platform- and language independent notation.
- A vast number of telecommunication protocols and services are defined using ASN.1. This means that pre-defined ASN.1 packages and modules are available and can be obtained from standardization organizations, RFCs, etc. For instance, the ASN.1 data types defining SNMP are available in RFC 1157.
- When ASN.1 data types are transmitted over computer networks, their values must be represented in bit-patterns. Encoding and decoding rules determining the bit-patterns are already defined for ASN.1. The SDL suite supports BER and PER encoding.
- ASN.1 enables extensibility. This means that it simplifies compatibility of systems that have been designed and implemented large time frames apart.
- The SDL suite and the TTCN suite can share common data types by specifying these in a separate ASN.1 module.

Implementation of ASN.1

When importing ASN.1 data types to your SDL system, you need to translate the ASN.1 definitions to SDL. The SDL suite does this for you using a tool called ASN.1 Utilities. This tool is automatically invoked when you analyze your SDL system.

However, having the ASN.1 data types translated to SDL is not enough to include them in your application. If you are going to transfer application-generated information on computer networks, the values of the

data types must be encoded. When transferring signals in or out of your SDL system, you must also create the interface between the environment and the system.

Thus, the process of implementing ASN.1 data types can be divided into three separate steps:

1. Creating the abstract syntax
2. Creating the transfer syntax
3. Compiling the application

The definitions of the abstract syntax and the transfer syntax is presented below.

Abstract Syntax

The basic idea is to transport some type of information between two nodes using protocol messages. The abstract syntax is defined as the set of all possible messages that can be transported. To create the abstract syntax you must:

- design some form of data structure defined in a high-level programming language, for instance ASN.1.
- define the possible set of values that the data structure can take.

Transfer Syntax

The transfer syntax is the set of bit patterns that represents the abstract syntax messages with each bit pattern representing just one value. The rules determining the bit-patterns are called the encoding rules.

Creating the Abstract Syntax

When creating the abstract syntax you must perform the following tasks:

- Adding ASN.1 modules to your project
- Importing the ASN.1 modules in your SDL diagrams.
- Assigning values to the data types.

Adding ASN.1 Modules to your Project

An ASN.1 module is a file containing the ASN.1 data type definitions. If you are implementing a standard communication protocol, it is very likely that pre-defined ASN.1 modules have been created. The modules can be obtained from standardization organizations, RFCs, etc. In Example 10 on page 323, the ASN.1 module that defines the SNMP protocol is presented. This module is available in the RFC 1157.

However, should a pre-defined module not be available for your type of application, you must create your own module. Please see adequate ASN.1 literature for instructions and guidelines on creating ASN.1 modules.

Regardless how you obtain the ASN.1 modules, you must add the module to your project before the SDL suite can include the data types.

Follow the instructions below to add the ASN.1 module to your project.

1. Save your ASN.1 module in a subdirectory to your project. Make sure that you append the **.asn** file extension to the saved module.
2. Open the Organizer and select the chapter where you want to include the module. This is done by clicking the chapter marker, for instance the *Other Documents* marker, see Figure 191 on page 301.

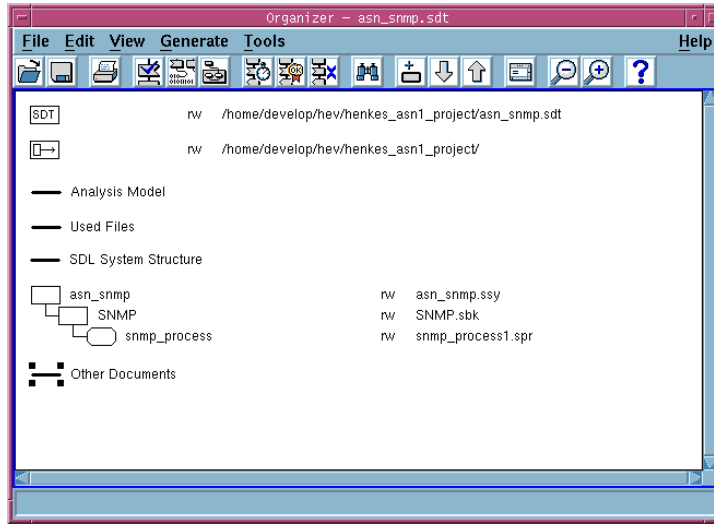

Figure 191: Selection of chapter marker

3. From the *Edit* menu, select the *Add Existing...* command. The Add Existing window opens.

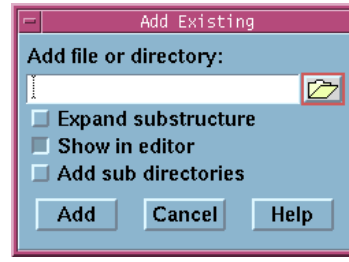

Figure 192: The Add Existing window

4. Click the folder image button in order to find your ASN.1 module. The Select file to add window opens.

5. Select the directory you want to search and change the search filter, by typing *.asn in the *Filter* field. Click the *Filter* button. The available ASN.1 modules are now displayed in the *Files* window. Select

module and click the *OK* button. The Select file to add window closes.

6. The selected module is now displayed in the Add Existing window. Just click the *OK* button to add the module to your system. The module should now be visible in the Organizer in your selected chapter.

The ASN.1 modules are now added to your project.

Example 1: Adding ASN.1 modules to SDL project ––––––––––––––

In the SNMP example, the three modules RFC1155_SMI, RFC1157_SNMP and USE_SNMP have been added to the project.

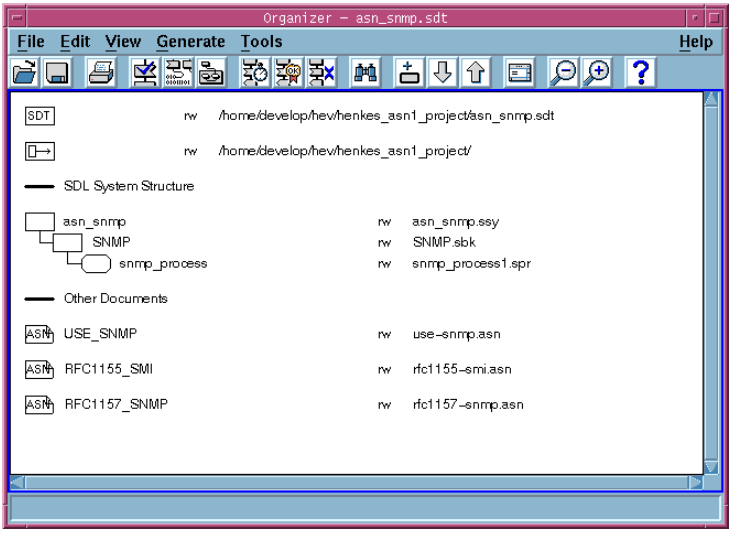

Figure 193: View of added ASN.1 modules

––

Importing ASN.1 Modules

After the modules have been added to the project, they must be made available to the SDL system. This is done by importing the modules to

the SDL system file. When the modules have been imported, the ASN.1 data types can be used as regular SDL types.

Follow the instructions below to import the modules in the SDL diagrams:

1. From the Organizer, open the system file, <system_name>.ssy.
2. Enter the name of the added modules in the package reference frame, which is located outside the system frame. (See Figure 194).
3. Save the diagram.

Example 2: Importing ASN.1 modules ––––––––––––––––––––––––––

In the SNMP example, the three modules RFC1155_SMI, RFC1157_SNMP and USE_SNMP are imported.

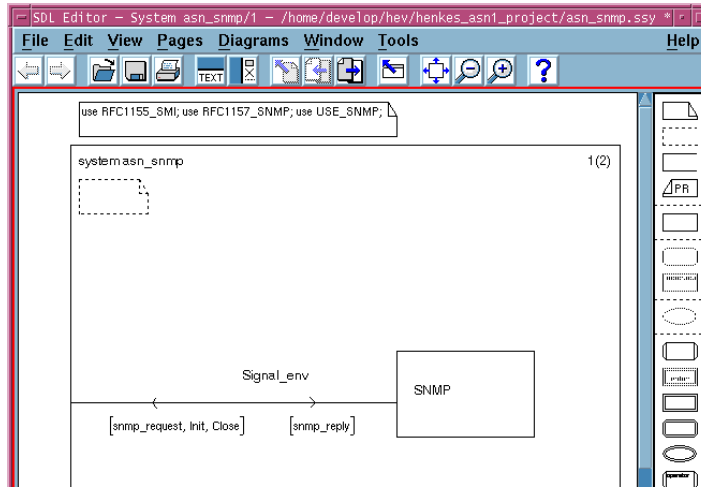

Figure 194: View of the imported ASN.1 modules

––

Assigning Values to the Data Types

When the modules are imported to the SDL system, you are free to declare signal parameters and variables of ASN.1 data types. The parameters and variables are treated as regular SDL parameters and variables, and you assign values to them in the same manner as you normally do.

When declaring signals that are transporting information defined using ASN.1, it is recommended that you define a top-level type of a ASN.1 module as the signal parameter.

When your variables have been assigned values, you have created the abstract syntax.

Example 3: Assigning values to variables ––––––––––––––––––––––

In this example the variable *Reply* has been declared as the type *Message*. This type is a top-level type that is defined in the ASN.1 module SNMP1157.asn, see Example 10 on page 323.

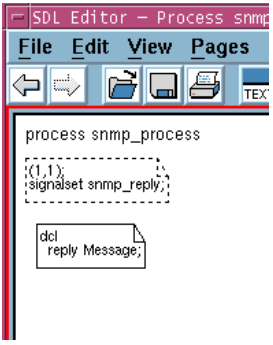

Figure 195: Declaration of the reply variable

When the variable `reply` has been declared, you can use it in the SDL diagram as a regular SDL variable. Figure 196 on page 305 shows how `reply` is used as the argument of a signal that has been received by the SDL system.

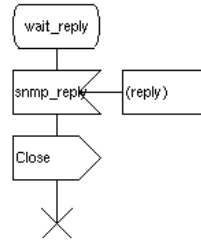

Figure 196: Usage of the `reply` *variable*

Example 4: Declaring signals ----------

The signal `snmp_reply` in the previous example must be declared before it can be used. As mentioned before, the argument of the signal is a top-level type of the ASN.1 module.

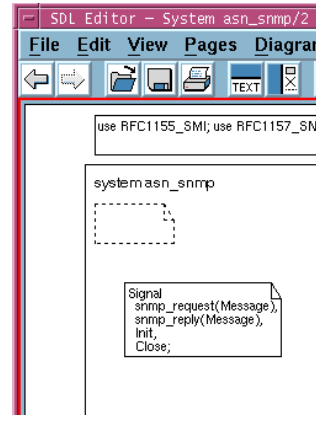

Figure 197: Signal Declarations

Creating the Transfer Syntax

The SDL suite offers several way to create the transfer syntax. The available coding access interfaces are:

- Basic SDL interface
- Extended SDL interface
- C code interface

In this tutorial, only the C code interface will be covered. For a complete description of ASN.1 encoding and decoding, please see chapter 59, *ASN.1 Encoding and De-coding in the SDL Suite, in the User's Manual*.

Using the C code interface, the transfer syntax can be created either using the Organizer's make dialog or using the Targeting Expert. Both methods are presented in this tutorial. When using the Targeting Expert, you can select to use the Cadvanced SDL to C Compiler or the Cmicro SDL to C Compiler when creating the transfer syntax. Both methods will be covered as well.

This section starts with a short introduction and the actual instructions are presented in:

- "Generating Template Files - the Organizer" on page 310
- "Generating Template Files - Targeting Expert" on page 315

Introduction

To be able to transfer the abstract syntax between two nodes in network, you must first create the transfer syntax. The transfer syntax representation is then transmitted in a protocol buffer.

When creating the transfer syntax you must perform the following tasks:

- Generating template files
- Editing the generated template files

The template files must be generated in order for you to include the ASN.1 data types in the compilation and code generation processes. The template files extract information from your SDL system and create a skeleton. Often these template files do not contain sufficient informa-

tion to meet the demands of the application and therefore you must edit the templates. The template files that are generated cover the following areas:

- the environment functions
- the make process

However, the SDL suite needs additional information in order to create the environment file. Before the generation you must determine which encoding/decoding schemes to use and you must create type nodes files.

Note: Environment File

There are several ways to create the environment file. This tutorial shows how to auto-generate the file. However, you can also make your own file from scratch. This procedure is more advanced and is only partially covered.

Note: Type Nodes

The type nodes are auto-created by ASN.1 Utilities and must not be edited.

Note: The Make process

The template makefile is only created if you are using the Make dialog. The default makefile of the Targeting Expert handles all necessary make functionality.

Environment Functions

The environment is defined as all devices or functions that are needed by the application but not specified within the SDL system. By sending signals to the environment, the SDL system wants certain tasks to be performed. This could be for instance:

- reading or writing information to a file
- sending or receiving messages across the network
- reading or writing information on hardware ports or sockets

However, the SDL system only controls events that occur within the system. It does not specify how signals leaving the system are handled by the environment. This is why you must provide an interface between

the SDL system and the environment. This interface is made up by the environment functions.

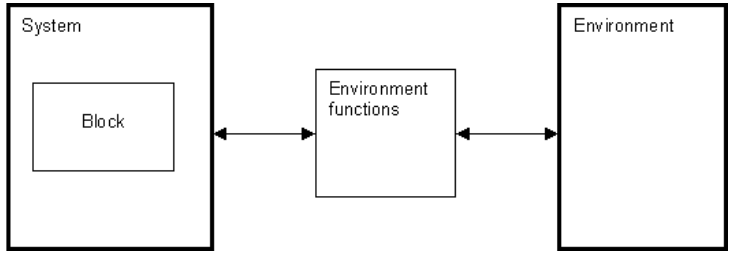

Figure 198: The environment functions

The SDL suite is rather helpful and can generate a template environment file that includes a skeleton of the environment functions. The environment file is written in C code and by editing this file you can specify the behavior of signals from the SDL system and of signals going in to the SDL system.

An environment header file or system interface header file can also be created. This file contains all type definitions and other external definitions that are necessary in order to implement the environment functions.

Note: Environment files

There are several ways to create the environment file. You can:

- auto-create the file. This procedure is covered in this tutorial.
- make your own file from scratch. This is a more advanced procedure and is only partially covered in this tutorial.

Encoding/Decoding

When creating the transfer syntax, the messages that will be transferred must be encoded and the incoming messages must be decoded. The type of encoding rules to apply is specified in the environment file. This means that the encoding/decoding function calls must be included in the environment file.

The SDL suite supports the standard BER and PER encoding/decoding schemes, but it also allows you to use a user specified encoding scheme. ASCII encoding is available in the SDL suite as well, but it does not support encoding of ASN.1 types.

Type Nodes

To include the ASN.1 data types in your application, they must be translated into a form that the SDL suite understands. Within the SDL suite, this translation is handled by the ASN.1 Utilities.

The ASN.1 Utilities tool is invoked automatically when the SDL system is analyzed and it allows you to:

- perform syntactic and semantic analysis of your ASN.1 modules
- generate SDL code from the ASN.1 modules
- generate type information for encoding and decoding using BER or PER

This means that when you are using the ASN.1 utilities, you create type nodes. A type node is a static variable that describes the properties and characteristics of an ASN.1 data type, including tag information needed by BER/PER encoders and decoders. The variable is named `yASN1_<type_name>`.

All nodes are generated in files named `<asn1module_name>_asn1coder.c` and declarations to access them in files named `<asn1module_name>_asn1coder.h`.

> **Note:**
> The type nodes are auto-created by ASN.1 Utilities and must not be edited.

Make Process

> **Note: Make dialog only**
> This section is only valid if you build and analyze your project using the Organizer's make dialog.

The default makefile in the SDL suite, determines the relationship between source files, header files, object files and libraries in your project.

However, the default makefile does not include the generated files in the make process. To include the environment files and the type node files in the make process, you must generate a template makefile that will be appended to the default makefile, see Figure 199. The template makefile can be generated by the SDL suite.

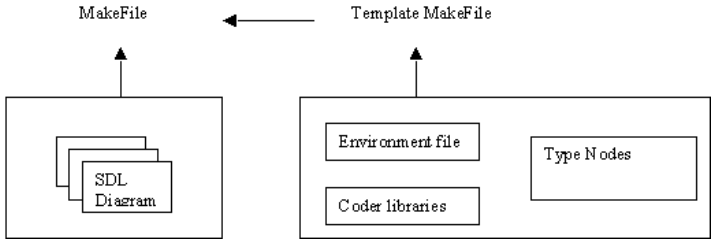

Figure 199: The make process

Generating Template Files - the Organizer

Follow the instructions below to generate environment files, type node files and the template makefile using the Organizer's Make dialog:

1. Click the SDL system symbol in the Organizer.
2. From the *Generate* menu, select the *Make...* command. The SDL Make window opens.
3. Specify your options in the make dialog according to the following list:
   - Select *Analyze & generate code*
   - From the *Code generator* drop-down list, select *Cadvanced*
   - Select *Generate environment header file*
   - Select *Generate environment functions*
   - Select *Generate ASN.1 coder*, to invoke ASN.1 Utilities.
   - From the *Use standard kernel* drop-down list, select *Application*

Note:

Make sure that you de-select the *Compile & link* option as you only want to generate the template files.

4. Specify your target directory where the generated files will be stored.

   Figure 200 shows the Make dialog with the selected options.

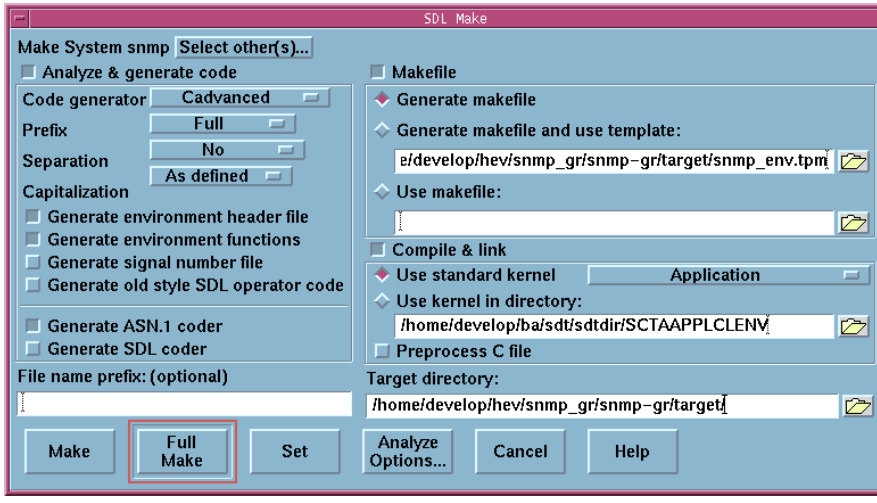

Figure 200: The Make dialog - generating template files

5. Press the *Full Make* button.

Note:

Encode and decode calls are only generated if the *Generate ASN.1 coder* option is enabled in the make dialog.

In your target directory, you will now find the generated files including:

- `<system_name>_env.c`
  This is the environment skeleton file.

- `<system_name>.ifc`
  This is the environment header file.

- **<system_name>_env.tpm**
  This is the template makefile.

- **<asn1module_name>_asn1coder.c**
  <asn1module_name>_asn1coder.h
  These files are the type nodes created by the ASN.1 utilities.

Editing the Generated Files - the Organizer

As the generated files only consist of skeleton functions, you must edit the files to suit the functionality of your application.

> **Note:**
>
> Make a habit of making a copy of the environment file and the template makefile after they have been edited. Otherwise the edits will be overwritten, if the files are re-generated from the Make dialog by mistake.

1. Edit the environment file `<system_name>_env.c` file using any text editor. In the skeleton file, macros are included but they are not defined. To define the macros, create a `<system_name>_env.h` file and enter the code manually. Example 5 on page 313 shows the unedited SNMP environment file.
2. Save the environment file.
3. Edit the template makefile `<system_name>_env.tpm` if necessary.
4. Save the template makefile.
5. Make copies of the edited files and save the copies in a different folder.

> **Notes:**
>
> - In order to transfer the information on the network, you must add socket commands to an appropriate header file.
> - If you want to use more than one encoding scheme, for instance BER and PER, you must enter the appropriate encoding function calls in the environment file

Example 5: Environment Functions ––––––––––––––––––––––––––

The following code is part of the environment file skeleton and displays the function that handles the out signals.

```
XENV_OUT_START


  /* Signals going to the env via the channel Signal_env */

    /* Signal snmp_request */
    IF_OUT_SIGNAL(snmp_request,"snmp_request")
      OUT_SIGNAL1(snmp_request,"snmp_request")
      XENV_BUF(BufInitWriteMode(Buf));
      XENV_ENC(BER_ENCODE(Buf, (tASN1TypeInfo *)&yASN1_Message,
        (void *)&((yPDef_snmp_request *)(*SignalOut))->Param1));
      OUT_SIGNAL2(snmp_request,"snmp_request")
      XENV_BUF(BufCloseWriteMode(Buf));
      RELEASE_SIGNAL
    END_IF_OUT_SIGNAL(snmp_request,"snmp_request")

    /* Signal Init */
    IF_OUT_SIGNAL(Init,"Init")
      OUT_SIGNAL1(Init,"Init")
      XENV_BUF(BufInitWriteMode(Buf));
      OUT_SIGNAL2(Init,"Init")
      XENV_BUF(BufCloseWriteMode(Buf));
      RELEASE_SIGNAL
    END_IF_OUT_SIGNAL(Init,"Init")

    /* Signal Close */
    IF_OUT_SIGNAL(Close,"Close")
      OUT_SIGNAL1(Close,"Close")
      XENV_BUF(BufInitWriteMode(Buf));
      OUT_SIGNAL2(Close,"Close")
      XENV_BUF(BufCloseWriteMode(Buf));
      RELEASE_SIGNAL
    END_IF_OUT_SIGNAL(Close,"Close")
}
```

––

Encoder and Decoder function calls

As stated earlier, it is not necessary to auto-create the environment file. By copying another environment file or by writing it from scratch, you can customize the environment file for your needs. If you do so you must use the correct syntax of the encoding and decoding functions.

The syntax of the BER function calls is:

```
BER_ENCODE (
   Buffer,
```

```
   &Typenode,
   &Signalparameter)

BER_DECODE (
   Buffer,
   &Typenode,
   &Signalparameter)
```

The syntax of the PER function calls is:

```
PER_ENCODE (
   Buffer,
   &Typenode,
   &Signalparameter)

PER_DECODE (
   Buffer,
   &Typenode,
   &Signalparameter)
```

Example 6: Encoding and Decoding function calls ––––––––––––––

The following function calls are being used in the SNMP example:

```
BER_ENCODE(Buf, (tASN1TypeInfo *)&yASN1_Message,
   (void *)&((yPDef_snmp_request *)(*SignalOut))-
>Param1));

BER_DECODE(Buf, (tASN1TypeInfo *)&yASN1_Message,
   (void *)&((yPDef_snmp_reply *)SignalIn)->Param1))
```

––

Decoding incoming signals

Before the SDL system can use the information that is encapsulated in the incoming signals, a number of tasks must be performed. Most of them are automatically performed by the SDL suite, but some must be handled manually.

The following list defines the steps involved in the decoding process. You must perform steps 1 and 2 manually, while steps 3 through 5 are generated by the SDL suite:

1. In order to auto-generate a correct environment file, declare incoming and outgoing signals in the SDL system. If signal parameters are declared as ASN.1 types, a top-level node in the ASN.1 module should be used, see Example 4 on page 305. It is recommended that

you use the same top-level type when decoding as you do when encoding.

2. Extract the encoded information from the protocol-specific packet and transfer it to a data buffer. This should be implemented in C code in the environment file. Instructions and further information is available in chapter 58, *Building an Application, in the User's Manual*

The functions and function calls of the following functionality are auto-generated in the environment file.

3. Memory for the signal structure is allocated.
4. The BER_DECODE function is called. The function is defined in the decoding library and handles the actual decoding process.
5. The decoded signal is sent to the SDL system. This is performed by the SDL_Output function.

Generating Template Files - Targeting Expert

Follow the instructions below to generate environment files, type node files and the template makefile using the Targeting Expert.

1. From the *Generate* menu, select the *Targeting Expert* command. The SDL Targeting Expert window opens.
2. From the drop-down menu located above the Partitioning Diagram Model frame, select *Light Integrations* and the desired SDL to C Compiler. It is possible to used either Cadvanced or Cmicro. The pre-defined alternative specifies the type of compiler needed for the generation.
3. Select the *SDL to C Compiler* tab.
4. In the *General* box, select *Analyze/generate code.*
5. In the *Environment* box, select:
   - *Environment functions*
   - *Environment header file*
6. Select the *Communication* tab. In the *Coders* box, select the *Generate ASN.1 coder functions* check box.

7. Press the *Full Make* button. This generates the environment file.

> **Note:**
> Encode and decode calls are only generated if the *Coder functions...* option is enabled.

In your target directory, you will now find the generated files including:

- **`<system_name>_env.c`** (Cadvanced)
  `env.c` (Cmicro)
  This is the environment skeleton file.
- **`<system_name>.ifc`**
  This is the environment header file.
- **`<asn1module_name>_asn1coder.c`**
  `<asn1module_name>_asn1coder.h`
  These files are the type nodes created by the ASN.1 utilities.

Editing the Generated Files - Targeting Expert

As the generated files only consist of skeleton functions, you must edit the files to suit the functionality of your application.

> **Note:**
> Make a habit of making a copy of the environment file after it has been edited. Otherwise the edits will be overwritten, if the file is re-generated by mistake.

1. Rename the environment file.
2. Edit the environment file `<system_name>_env.c` file according to your needs. In the skeleton file, macros are included but they are not defined. To define the macros, create a `<system_name>_env.h` file and enter the code manually. Example 7 on page 317 shows the unedited SNMP environment file.
3. Save the environment file.

Notes:

- In order to transfer the information on the network, you must add socket commands to an appropriate header file.
- If you want to use more than one encoding scheme, for instance BER and PER, you must enter the appropriate encoding function calls in the environment file

Example 7: Environment Functions - Cadvanced ----------

The following code is part of the environment file skeleton and displays the function that handles of the out signals.

```
XENV_OUT_START


  /* Signals going to the env via the channel Signal_env */

    /* Signal snmp_request */
    IF_OUT_SIGNAL(snmp_request,"snmp_request")
      OUT_SIGNAL1(snmp_request,"snmp_request")
      XENV_BUF(BufInitWriteMode(Buf));
      XENV_ENC(BER_ENCODE(Buf, (tASN1TypeInfo *)&yASN1_Message,
        (void *)&((yPDef_snmp_request *)(*SignalOut))->Param1));
      OUT_SIGNAL2(snmp_request,"snmp_request")
      XENV_BUF(BufCloseWriteMode(Buf));
      RELEASE_SIGNAL
    END_IF_OUT_SIGNAL(snmp_request,"snmp_request")

    /* Signal Init */
    IF_OUT_SIGNAL(Init,"Init")
      OUT_SIGNAL1(Init,"Init")
      XENV_BUF(BufInitWriteMode(Buf));
      OUT_SIGNAL2(Init,"Init")
      XENV_BUF(BufCloseWriteMode(Buf));
      RELEASE_SIGNAL
    END_IF_OUT_SIGNAL(Init,"Init")

    /* Signal Close */
    IF_OUT_SIGNAL(Close,"Close")
      OUT_SIGNAL1(Close,"Close")
      XENV_BUF(BufInitWriteMode(Buf));
      OUT_SIGNAL2(Close,"Close")
      XENV_BUF(BufCloseWriteMode(Buf));
      RELEASE_SIGNAL
    END_IF_OUT_SIGNAL(Close,"Close")
}
```

Example 8: Environment functions - Cmicro ––––––––––––––––––––

The following code is part of the environment file skeleton and displays the function that handles of the out signals.

```
  switch (xmk_TmpSignalID)
  {
     case snmp_request :
          {
            /* BEGIN User Code */
            /* Use (yPDP_snmp_request)xmk_TmpDataPtr to access the signal's
parameters */
            /* ATTENTION: the data needs to be copied. Otherwise it */
            /*            will be lost when leaving xOutEnv */
            /* This section can be used to encode outgoing data with the
selected coder functions.
            ** Please remove the comments and send the data with your
communications interface!
            ** (<SendViaCommunicationsInterface( data, datalen )> must be
replaced)
            char* data;
            int datalen;

            BufInitWriteMode( Buf );
            XENV_ENC( PER_ENCODE( Buf, (tASN1TypeInfo *)
&yASN1_z_RFC1157_SNMP_0_Message,
                           (void *) &((yPDef_snmp_request *)xmk_TmpDataPtr)-
>Param1));
            BufCloseWriteMode( Buf );
            BufInitReadMode( Buf );
            datalen = BufGetDataLen(Buf);
            data = BufGetSeg( Buf, datalen );
            <SendViaCommunicationsInterface( data, datalen )>;
            BufCloseReadMode( Buf );
            */
            /* Do your environment actions here. */
            xmk_result = XMK_TRUE; /* to tell the caller that */
                                   /* signal is consumed      */
            /*   END User Code */
          }
          break ;

     case Init :
          {
            /* BEGIN User Code */
            /* Do your environment actions here. */
            xmk_result = XMK_TRUE; /* to tell the caller that */
                                   /* signal is consumed      */
            /*   END User Code */
          }
          break ;

     case Close :
          {
            /* BEGIN User Code */
            /* Do your environment actions here. */
            xmk_result = XMK_TRUE; /* to tell the caller that */
                                   /* signal is consumed      */
            /*   END User Code */
          }
          break ;
```

```
   default :
        xmk_result = XMK_FALSE;/* to tell the caller that */
                               /* signal is NOT consumed  */
                               /* and to be handled by    */
                               /* the Cmicro Kernel ...   */
        break ;
}
```

Encoder and Decoder function calls

As stated earlier, it is not necessary to auto-create the environment file. By copying another environment file or by writing it from scratch, you can customize the environment file for your needs. If you do so you must use the correct syntax of the encoding and decoding functions.

The syntax of the BER function calls is:

```
BER_ENCODE (
   Buffer,
   &Typenode,
   &Signalparameter)

BER_DECODE (
   Buffer,
   &Typenode,
   &Signalparameter)
```

The syntax of the PER function calls is:

```
PER_ENCODE (
   Buffer,
   &Typenode,
   &Signalparameter)

PER_DECODE (
   Buffer,
   &Typenode,
   &Signalparameter)
```

Example 9: Encoding and Decoding function calls ----------

The following function calls are being used in the SNMP example:

```
BER_ENCODE(Buf, (tASN1TypeInfo *)&yASN1_Message,
   (void *)&((yPDef_snmp_request *)(*SignalOut))-
>Param1));

BER_DECODE(Buf, (tASN1TypeInfo *)&yASN1_Message,
   (void *)&((yPDef_snmp_reply *)SignalIn)->Param1))
```

Decoding incoming signals

Before the SDL system can use the information that is encapsulated in the incoming signals, a number of tasks must be performed. Most of them are automatically performed by the SDL suite, but some must be handled manually.

The following list defines the steps involved in the decoding process. You must perform steps 1 and 2 manually, while steps 3 through 5 are generated by the SDL suite:

1. In order to auto-generate a correct environment file, declare incoming and outgoing signals in the SDL system. If signal parameters are declared as ASN.1 types, a top-level type in the ASN.1 module should be used, see Example 4 on page 305. It is recommended that you use the same top-level type when decoding as you do when encoding.

2. Extract the encoded information from the protocol-specific packet and transfer it to a data buffer. This should be implemented in C code in the environment file. Instructions and further information is available in chapter 58, *Building an Application, in the User's Manual*.

The functions and function calls of the following functionality are auto-generated in the environment file.

3. Memory for the signal structure is allocated.

4. The BER_DECODE function is called. The function is defined in the decoding library and handles the actual decoding process.

5. The decoded signal is sent to the SDL system. This is performed by the SDL_Output function.

Compiling Your Application

After you have created the transfer syntax, you are ready to compile and build your application, including the edited environment file and the template makefile. Follow the appropriate instructions as presented in:

- "Using the edited files - Organizer" on page 321
- "Using the edited files - Targeting Expert" on page 322

Using the edited files - Organizer

Please follow the instructions below:

1. Click the SDL system symbol in the Organizer
2. From the *Generate* menu, select the *Make...* command. The SDL Make window opens.
3. Change your options in the make dialog according to 0 following list:
   - Select *Analyze & generate code*
   - From the *Code generator* drop-down list, select *Cadvanced*
   - De-select *Generate environment header file*
   - De-select *Generate environment functions*
   - De-select *Generate ASN.1 coder*
   - Select the *Makefile* button
   - Select *Generate makefile and use template* and enter the generated template makefile `.... /<new_name>_env.tmp` in the text field.
   - Select *Compile & link*
   - From the *Use standard kernel* drop-down list, select *Application*.

   Figure 201 on page 322 shows the Make dialog with the selected options.

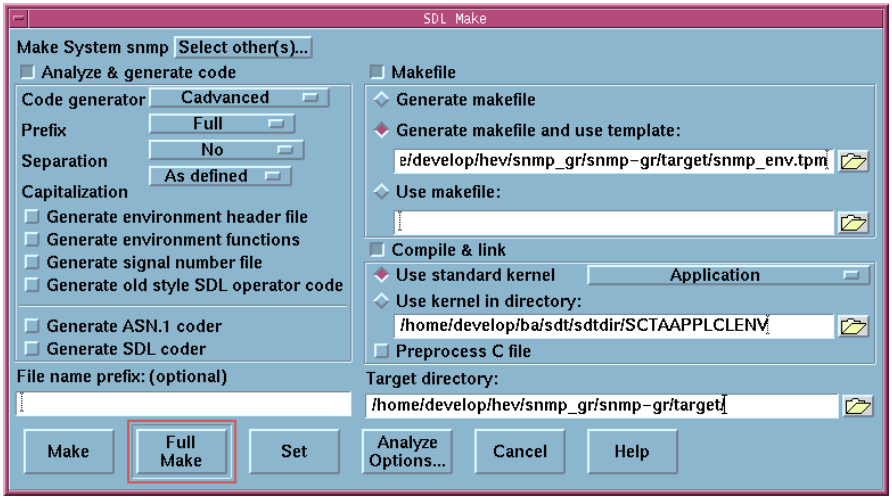

Figure 201: The Make dialog - Compiling

4. Specify your target directory where the generated files will be stored.

5. Press the *Full Make* button.

Using the edited files - Targeting Expert

Please follow the instructions below:

1. Select the *SDL to C Compiler* tab.

2. In the *Environment* box, de-select the *Environment functions* option and the *Environment header file* option.

3. Press the *Full Make* button.

Appendix A

Example 10: The RFC1157 ASN.1 Module ––––––––––––––––––––––

```
RFC1157-SNMP DEFINITIONS ::= BEGIN

IMPORTS
  ObjectName, ObjectSyntax, NetworkAddress, IpAddress, TimeTicks
            FROM RFC1155-SMI;

-- top-level message

        Message ::=
                SEQUENCE {
                     version        -- version-1 for this RFC
                        INTEGER {
                            version-1(0)
                        },

                    community      -- community name
                        OCTET STRING,

                    data           -- e.g., PDUs if trivial
                        PDUs --ANY--        -- authentication is being used
                }

-- protocol data units

        PDUs ::=
                CHOICE {
                    get-request
                        GetRequest-PDU,

                    get-next-request
                        GetNextRequest-PDU,

                    get-response
                        GetResponse-PDU,

                    set-request
                        SetRequest-PDU,

                    trap
                        Trap-PDU
                     }

                  GetRequest-PDU ::=
                      [0]
                          IMPLICIT SEQUENCE {
                              request-id
                                  RequestID,

                              error-status        -- always 0
                                  ErrorStatus,

                              error-index         -- always 0
                                  ErrorIndex,

                              variable-bindings
                                  VarBindList
                          }
                  GetNextRequest-PDU ::=
                      [1]
```

```
                    IMPLICIT SEQUENCE {
                        request-id
                            RequestID,
                        error-status        -- always 0
                            ErrorStatus,

                        error-index         -- always 0
                            ErrorIndex,

                        variable-bindings
                            VarBindList
                    }
                GetResponse-PDU ::=
                    [2]
                        IMPLICIT SEQUENCE {
                        request-id
                            RequestID,
                        error-status
                            ErrorStatus,

                        error-index
                            ErrorIndex,

                        variable-bindings
                            VarBindList
                        }
                SetRequest-PDU ::=
                    [3]
                        IMPLICIT SEQUENCE {
                        request-id
                            RequestID,

                        error-status        -- always 0
                            ErrorStatus,

                        error-index         -- always 0
                            ErrorIndex,

                        variable-bindings
                            VarBindList
                        }

Trap-PDU ::=
    [4]

        IMPLICIT SEQUENCE {
           enterprise           -- type of object generating
                                -- trap, see sysObjectID in [5]
               OBJECT IDENTIFIER,

           agent-addr           -- address of object generating
               NetworkAddress, -- trap

           generic-trap         -- generic trap type
               INTEGER {
                   coldStart(0),
                   warmStart(1),
                   linkDown(2),
                   linkUp(3),
                   authenticationFailure(4),
                   egpNeighborLoss(5),
                   enterpriseSpecific(6)
               },

           specific-trap     -- specific code, present even
               INTEGER,      -- if generic-trap is not
```

```
                                -- enterpriseSpecific

                    time-stamp          -- time elapsed between the last
                      TimeTicks,        -- (re)initialization of the network
                                        -- entity and the generation of the trap

                    variable-bindings   -- "interesting" information
                         VarBindList
                }

                    -- request/response information
                     RequestID ::=
                             INTEGER

                     ErrorStatus ::=
                             INTEGER {
                                 noError(0),
                                 tooBig(1),
                                 noSuchName(2),
                                 badValue(3),
                                 readOnly(4),
                                 genErr(5)
                             }

                     ErrorIndex ::=
                             INTEGER

                     -- variable bindings

                     VarBind ::=
                             SEQUENCE {
                                 name
                                     ObjectName,

                                 value
                                     ObjectSyntax
                             }

                     VarBindList ::=
                             SEQUENCE OF
                                 VarBind

END
```

Chapter 8 Tutorial: Using ASN.1 Data Types