Chapter

57 *The Cadvanced/Cbasic SDL to C Compiler*

The Cadvanced/Cbasic SDL to C Compiler translates your SDL system into a C program that you can compile and link together with a runtime library to form an executable program such as a simulator, a validator or, in the case of Cadvanced, an application.

This chapter is a reference manual to the Cadvanced/Cbasic SDL to C Compiler. There are also a number of other chapters related to code generation:

- **In *chapter 50, The SDL Simulator*, you will find a reference to the simulation facilities in the SDL suite. In chapter 51, *Simulating a System*, you will find a user's guide to the simulator.**
- **In *chapter 53, The SDL Validator*, you will find a reference to the validation facilities in the SDL suite. In chapter 54, *Validating a System*, you will find a user's guide to the validator.**
- **In chapter 58, *Building an Application*, you may read about how to generate applications for host and target environments.**
- **In chapter 62, *The Master Library*, you will find information about how to customize your own libraries for a specific purpose, such as application generation for target computers. The chapter also describes the structure of the generated C code and the internal data structures in the generated C code.**
- **In chapter 63, *The ADT Library*, you will find a reference to the library of abstract data types that is distributed with the SDL suite and that you may use in your systems designed in SDL. Some examples of use are also available.**
- **In chapter 64, *The Performance Library*, you may read about how to generate and run simulators which are specially adapted for the area of performance simulation.**

Introduction

Application Areas for the Cadvanced/Cbasic SDL to C Compiler

There are a number of application areas for the Cadvanced/Cbasic SDL to C Compiler, for example:

- Functional simulation and debugging of protocol specifications
- Debugging of system designs described in SDL
- Generation of applications, including embedded system applications with real time characteristics
- Performance simulations
- Simulation of the behavior behind a user interface prototype

In this part of the manual, the general behavior of the code generator and its application for simulation and debugging are discussed. The possibility to generate simulators is described in chapter 50, *The SDL Simulator*.

Functional Simulation and Debugging

During the validation of a specification or design of an application expressed in ITU SDL, you can use the Cadvanced/Cbasic SDL to C Compiler as a tool for simulation to help you understand and debug the behavior of a system description. (See *chapter 50, The SDL Simulator*.)

Errors arising from two different areas have to be considered in the validation process. In the language domain, errors due to illegal or illogical usage of the language concepts might be introduced into the specification; while in the problem domain, logical errors might be introduced.

With traditional computer program development, most illegal uses of language concepts are found by compilers or by run-time systems. Examples are syntax errors, missing declarations, division by zero, or indexing an array out of its bounds.

In the problem domain, however, the only feasible ways of detecting logical errors in non-trivial programs are testing and proofreading. When it comes to specifications in SDL, language domain errors can be

detected by using the SDL Analyzer, which can be seen as a compiler without a code generation facility (see *chapter 55, The SDL Analyzer*). To detect problem domain errors, testing by simulating the specification is the main procedure available. Please see also chapter 53, *The SDL Validator*.

The specification of a protocol in SDL, for instance, specifies a signal interface by giving a hypothetical implementation of the components in the protocol. This strategy immediately brings up two different purposes for simulating the behavior of a system specification: to understand the external view and to understand the internal view.

In the external view, the signal interface is of concern, while the internal behavior of the system specification (the behavior of the processes in the system) is of little or no interest. In the internal view, the internal behavior of the system specification is of concern, while the external signal interface is simply seen as part of the internal behavior.

A simulation of the internal behavior of a system specification constitutes an important part of the validation of the specification, both as a debugging tool and as a means to increase the understanding of the dynamic behavior of the specification. A designer of a system might use this kind of simulation to understand the specification better.

The ability to simulate and debug applications generated by the code generator at an SDL level is a very important feature towards achieving the correct overall behavior of the application. The debugging facilities provided by the SDL suite have much in common with interactive debuggers for ordinary programming languages. The debugging is performed on a host computer.

Another application of the code generator as a simulator generator, is of course in SDL education, where simulation, especially of the internal behavior of a system specification, can serve as a powerful way of clarifying the semantics of SDL concepts.

Performance Simulation

The Cadvanced/Cbasic SDL to C Compiler can be used for performance simulations. You describe the performance model of the actual system using SDL. This model can be translated to a simulation and executed. By introducing measurements of interesting data, such as queue lengths, delays, and so on, into the SDL model, it is possible to gather statistical data during the execution of the simulation. In chapter 64, *The*

Performance Library, you can find a description of the performance simulation facilities.

To simplify this kind of simulation, a number of SDL abstract data types and their implementations have been developed, where, for example, random number generation and handling of queues are supported. Please see chapter 63, *The ADT Library*.

Validation

The SDL Validator uses the code produced by the Cadvanced/Cbasic SDL to C Compiler to form a program suitable for validation of an SDL system. The Validator uses state space exploration and can be used to:

- Find run time errors
- Verify MSCs against the SDL system
- Verify user defined rules

The Validator is described in *chapter 53, The SDL Validator*.

Communicating Simulations

You can specify that a generated C program should be able to communicate over the *PostMaster*, which is the mechanism used for communication between the SDL suite tools. Signals sent from the SDL system (the generated program) to the environment and signals coming to the SDL system from the environment can be handled. This facility makes it, for example, possible to develop simulation programs for two communicating systems, execute them using the SDL suite and obtain communication between the systems.

As a generated C program does not know what it communicates with, it can of course communicate with any type of application, as long as the application is connected to the *PostMaster* (the communication medium) and sends signals according to the defined format. How to achieve this is described in *chapter 13, Using the Telelogic Tau Public Interface*.

A very interesting group of such applications are user interfaces. By connecting a user interface and an SDL simulation you can achieve several things: You can, for example, build well-designed application oriented user interfaces that present what is going on in a simulation, or you can in a simple way define the logic behavior behind a user interface during its prototyping phase.

Overview of the Cadvanced/Cbasic SDL to C Compiler

To facilitate the validation of SDL specifications or descriptions, the SDL Analyzer contains an SDL parser, an SDL semantic checker, and the Cadvanced/Cbasic SDL to C Compiler.

Creating a C Program

To obtain an executable program that behaves according to an SDL description, you enter the SDL description into the SDL Analyzer, which contains the Cadvanced/Cbasic SDL to C Compiler. If the SDL description is syntactically and semantically correct, a C program is generated. You then compile this program using an ordinary C compiler and link it with a predefined SDL run-time library to form an executable program. See Figure 489.

As indicated above, the C code generation facility contains two components:

- The SDL to C Compiler, which can be seen as a back-end to the SDL Analyzer. This component generates a C program.
- Predefined and precompiled C units, which implement an **SDL runtime library** and the command line user interface of a simulator, that is, a **monitor system**. The run-time library also includes a **communication mechanism** which makes it possible to trace the execution of SDL transitions in the SDL Editor. There are several versions of the library that are suitable to different application areas for the generated C code, see "Libraries" on page 2698 in chapter 58, *Building an Application*.

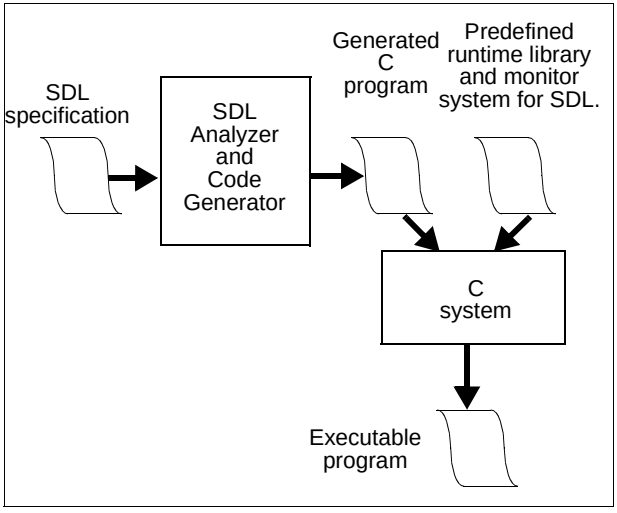

Figure 489: The production of an executable program

Executing a C Program as a Simulator

The generated C program uses an internal data representation of the SDL objects in the system, for example process instances and signal instances. The process instances will execute transitions in a quasi-parallel manner. During a transition, SDL actions such as tasks, decisions and signal outputs are executed according to the semantics of SDL.

You interact, using SDL terminology, with the simulator through a monitor system, which contains a number of commands to:

- Control the execution of transitions.
- Examine the status of objects in the system.
- Turn log facilities on and off.
- Affect the system by, for example, sending signal instances from the environment to the system.

Contents of This Chapter

You can find more details on creating and executing a C program in the following sections:

- In "Generating a C Program" on page 2568, the process of generating a C program is described.
- In "Abstract Data Types" on page 2586, implementation aspects especially concerning abstract data types, are described.
- In "Directives to the Cadvanced/Cbasic SDL to C Compiler" on page 2649, possibilities to give additional information to the Cadvanced/Cbasic SDL to C Compiler are discussed.
- The section "Using Cadvanced/Cbasic SDL to C Compiler to Generate C++" on page 2677, contains considerations on class definitions as C++ code and the utilization of the classes as C++ code in SDL tasks.
- In "Restrictions" on page 2681, the Cadvanced/Cbasic SDL to C Compiler restrictions are covered.

Generating a C Program

A generated C program can be used for several things, for example as a simulator, as a validator, or as an application with the behavior defined by the translated SDL system. The process of generation of simulators, validators, or applications, see below, is started in the Organizer, in the make dialog (see "Make" on page 119 in chapter 2, *The Organizer*) or by the quick buttons for simulation and validation (see "Quick Buttons" on page 176 in chapter 2, *The Organizer*).

The SDL Analyzer, which contains the C code generation facility, can also be started as a stand-alone tool. For more information about this possibility please see "The Analyzer Command-Line UI" on page 2404 in chapter 55, *The SDL Analyzer*.

Process of Generating a C Program

There are four steps that must be performed to start the execution of, for example, a simulator:

1. The SDL Analyzer and its built-in Cadvanced/Cbasic SDL to C Compiler create a program expressed in C source code.
2. The generated C file (or files) is compiled.
3. The compiled file (or files) is linked together with a predefined library.
4. The executable program that is created in the link operation is started.

This process has been automated and requires no user knowledge about compiling or linking of programs. The process is initiated in the Organizer using the quick buttons for simulation and validation, or by using the *Make* dialog.

A C program can only be generated for an SDL system. The C code that constitutes the program can, however, be generated on multiple files, which means that a local change in, for example, a block diagram only requires a regeneration and recompilation of the code for that unit. The object files, (the compiled versions of the C files) for the other unchanged units can then be used in the link operation to form a new executable program. This feature in automatically used by the make facility and the quick buttons, to minimize the amount of work and thus the turn-around time, for the process from a change in the SDL system to a new simulator (for example).

The separation of the C code for an SDL system can be decided by the user. The *Edit Separation* command from the *Generate* menu is used for that purpose, see "Edit Separation" on page 136 in chapter 2, *The Organizer*. The effect on the generated file structure and some guidelines of how to use separation can be found in the section "Selecting File Structure for Generated Code – Directive #SEPARATE" on page 2650.

Executing a C Program

The generated C programs can in principle be compiled as either a simulator, a validator, or an application. Generated applications have no further connection with the SDL suite and are executed as any other application.

A generated simulator or validator can however be started in two different ways:

- From the *Simulator Graphical User Interface*, which is started from the Organizer with the *SDL > Simulator UI* command and provides a graphical interface with buttons and menus and of course full connection to other SDL suite tools. See *"Graphical User Interface" on page 2130 in chapter 50, The SDL Simulator* for more information.
- From an OS shell, just like any other executable program. The user then invokes a *command line monitor* system. If the Organizer is running when starting a simulator or validator, the program will connect itself to the SDL suite. If the Organizer is not running or the simulator/validator is started with the program parameter `-nosdt`, the program will not connect itself to the SDL suite.

The SDL Unit for Which Code is Generated

The first time a C program is generated for a system, the complete system will be selected for analysis and C code generation. After that only the unit (*system* or *block*) that is changed will be selected. Note that the lowest level of possible regeneration object is a block. That block may not be a block type, or be part of a block type or system type. The reason that a process cannot be generated without regenerating the enclosing block, is that internal process information about, for example, formal parameters are used to generate code for other processes within the same block.

- Only complete C files can be generated. If, for example, the user has specified that a *block* and a *sub-block* should be generated on the

same file, it is not possible to regenerate code only for the sub-block.

- If the file structure is changed (by, for example, changes in the *Edit Separation* command or in the #SEPARATE directives), then the complete system is regenerated.

Errors During Code Generation

Errors that may occur during code generation are internal errors. That is, errors due to not yet implemented features of SDL, and errors related to problems with opened or closed operations of files.

An error message starts with an SDT references and is followed by a description of the error, including a error number. Example:

```
ERROR 884 Not implemented: Signal refinement
```

Features

Partitioning

General Ideas

The partitioning concept is a way to divide one SDL system into several applications. As a special case this means also that it is possible to simulate and validate selected parts of a system. You should note the difference between partitioning and separation. The partitioning feature is a way to select the parts of an SDL system which should be handled, while the separation feature is a way to select the file structure for the generated files.

To select a partition (or a program) it is, in simple cases, possible to use selections in the Organizer, and in more general cases possible to work with *build scripts*, i.e. text files containing commands to the Analyzer (the syntax used when running the Analyzer stand-alone). The restriction in the Organizer view is that only one selection can be handled and that instantiation of OO types cannot be selected. In a build script on the other hand, several component commands can be used to select several parts of a system. As the component command takes an SDL qualifier as a parameter, instantiations can also easily be selected.

Using Selections in the Organizer

To start with the simple case when one block or process should be simulated, this is easy to perform directly from the Organizer: Select the proper block or process and press the *Simulate* button (or go via *Make* dialog).

> **Note:**
>
> If you already have generated a simulation from the Organizer, and want to generate a new one with other options or with another selection, you should perform a Full Make, as changes in options or selection is not handled by the build process. Otherwise compilation or link errors might be the result of the build process.

Only the system, a block or a process can be selected for simulation. Types, including procedures, are only definitions and are not executable objects, while services depend on its enclosing process and cannot be

simulated on its own because of these dependencies. Block and process instantiations can be simulated, but only using build scripts, as such objects cannot be selected in the Organizer. The discussion above is of course also valid for generation of validators and applications.

Unconnected Diagrams

As another special case, there might be unconnected diagrams in the Organizer, i.e. objects not bound to a file. If such an object is a block, process, or a procedure, C code can anyhow be generated resulting in, for example, a Simulator or Validator.

- If a block is unconnected, this is treated as an implicit partitioning excluding this block.
- If a process is unconnected, this is treated as an implicit partitioning excluding this process. If other processes try to “create” such a process, this will become a null-action just indicated in the textual trace. In an application, such a create action will cause a compilation error.
- If a procedure is unconnected, any call to this procedure is just indicated in the textual trace. In an application such a procedure call will cause a compilation error.

Build Scripts

In general cases, build scripts should be used to specify the build process. Using such a file there are a number of features that can be used.

- It is possible to generate code for several partitions, using independent options and potentially different code generators for different parts of the system, all in one build process.
- Each partition can consist of several objects, and objects might be instantiations.

There are two Analyzer commands, see *“Description of Analyzer Commands” on page 2406 in chapter 55, The SDL Analyzer*, that are of major interest for specifying a partition. First we have the Program command, which takes a name as parameter. Second we have the Component command, which takes a qualifier as a parameter. The Program command gives the start of a partitioning specification, while the Component command is used to select an SDL component that should be

part of the partitioning. A partition specification can of course contain a number of components. The Component command is very similar to a selection in the Organizer when running directly from the Organizer.

A program section in a build script typically starts with a Program command and ends with a Generate command.

Example 336: Build script ––––––––––––––––––––––––––––––––––

```
program MyExample
component system example/block b1
component system example/block b2/process p22
target-directory /home/jk/example/target
set-env-header on
set-modularity user
generate-advanced-c
```

In Windows, the target-directory command could, for example, be:

```
target-directory c:\example\target
```

––

The example above means that a program containing the implementation of the complete block b1 and the process p22 in block b2 is generated with the Cadvanced SDL to C Compiler. The modularity is user defined and a system header file (`.ifc` file) will also be generated.

Code from the code generators will be placed in a subdirectory with the name given in the Program command, to the directory given by target directory. If this subdirectory does not exist it will be created.

> **Note:**
>
> You should always include a target-directory command in a build script, as otherwise the target directory will depend on where the SDL suite is started!

In the example above the generated C code can be found in the directory `/home/jk/example/target/MyExample`
(**In Windows** `c:\example\target\MyExample`).

In the example below three programs are generated for three different partitionings, also using different code generators.

Example 337: Build script with several programs ––––––––––––––––––

```
target-directory /home/jk/example/target

program MyExample
component system example/block b1
component system example/block b2/process p22
set-env-header on
set-modularity user
set-kernel SCTDEBCOM
generate-advanced-c

program MyExample1
component system example/block b2/process p21
generate-micro-c

program MyExample2
component system example/block b3
set-modularity no
generate-chipsy-chill
```

––

Analyzer commands that are of the type “set up an option” can be placed outside of the Program commands. The options actually used at the generate commands, are the options set up after executing all the commands up to the generate command. All the possibilities in the Make dialog and the Analyze dialog in the Organizer are also provided as commands in the Analyzer. Please see *“The Analyzer Command-Line UI” on page 2404 in chapter 55, The SDL Analyzer*, for a list of all commands.

Note:

When build scripts are used, all features in the Analyzer will have its hard coded defaults, if it is not set in the build script. Preferences and your settings in the Organizer are ignored. The default values are given in *“The Analyzer Command-Line UI” on page 2404 in chapter 55, The SDL Analyzer*, or you can start a stand-alone analyzer (sdtsan) directly in an OS shell and issue the commands Show-Analyze-Options and Show-Generate-Options.

See also “SDL Make” on page 120 in chapter 2, *The Organizer*, for handling of build scripts in the Organizer.

Behavior of Generated Partitioning

The basic idea is to redirect all channel going to objects not part of the current partitioning to the environment. This operation is performed by the code generator at code generation time. This means that all signals sent between objects in the partitioning and objects outside the partitioning, will be seen as signals to or from the environment. This is true everywhere, in simulations, validations, applications, in generated environment header files (.ifc files), and in generated environment functions.

Generation of Support Files

The Cadvanced/Cbasic SDL to C Compiler can generate a number of support files, together with the ordinary `.c`, `.h`, and makefiles. These files are

- System header file (`.ifc`)
- Skeleton to environment functions (`_env.c`)
- Signal number file (`.hs`)
- Coder/decoder framework files (`_cod.c`, `_cod.h`)

The generation of these files can be selected in the Organizer Make dialog, or as an Analyzer command, depending on which interface is used. The details on the system header files and the environment function can be found in "The Environment Functions" on page 2702, while the signal number file can be used to assign numbers to all signals in the system. Signal number files are most used in connection with OS integrations.

Implementation

In this section some implementation details are presented, that can be useful for understanding how a generated simulation or application behaves. Abstract data types are treated in the next section.

Time

A generated C program can be executed in two modes with respect to the treatment of time:

- *Simulated time*
- *Real time*

Simulated Time

Using simulated time, which is the most useful mode for simulations, means that the time in the simulation has no connection with the wall clock. Instead the *discrete event simulation* technique is used. This technique is based on the idea that the current value for the simulation time (Now in SDL) is equal to the time at which the currently executing event is scheduled. After one event is finished, the simulation time is increased to the time when the next event is scheduled and this event is started. Events in SDL will be process transitions, timer outputs, and signals sent to the system from the environment. As an example, the use of the discrete event simulation technique means that if the next event is a timer output scheduled one hour from now, and the next transition is allowed to execute, the timer output will occur immediately. The simulation time will be increased by one hour, but the user does not have to wait one hour.

Real Time

If real time is used, then there will be a connection between the clock in the executing program and the wall clock. In the example above the user would have to wait one hour until the timer output took place. To implement real time a clock function provided by the operating system is used. Not all systems are suitable to simulate in this way. The time scale in the system ought to be seconds or maybe minutes, not milliseconds and not hours.

At program start up the system time, SDL Now, is zero. The system clock is stopped during the time the program spends in the monitor system.

Note:

The C standard function `time` used as real time clock returns the time in seconds. It does not handle parts of seconds. The implementation of the clock can be changed by re-implementing the function SDL_Clock in `sctos.c`.

Scheduling

The process instances in the simulated system will execute transitions that consist of actions like tasks, decisions, outputs, procedure calls, etc., according to the rules of SDL. It is assumed that a transition takes no time and that a signal instance is immediately placed in the input port of the receiver when an output operation occurs.

A transition is always executed without any interrupts, if the user does not manually rearrange the ready queue using an appropriate command provided by the monitor system (Rearrange-Ready-Queue). It is possible to execute a few SDL symbols in one transition and then to re-arrange the ready queue and execute another transition. The interrupted transition can afterwards be executed to its end.

A quasi-parallel strategy for selecting transitions to be executed is thus the basic scheduling mechanism. SDL does not in itself define an execution strategy so the selected strategy is therefore an allowed, but not the only, possible strategy for execution.

As a consequence of the execution strategy, a generated simulator is not directly suited for simulation of “timing effects”, that is, situations where the time or order of actions in different process instances is of vital importance.

Example 338: Scheduling ––––––––––––––––––––––––––––––––––

An example of such a situation is: Suppose a process instance A outputs two signal instances during the same transition, one to process instance B and one to process instance C. During the corresponding transitions of B and C, a signal instance is sent to process instance D.

If the behavior of the system is dependent on the order in which the signal instances are received in the input port of D, this is a hazard situation

where the execution speed of process instances and the delay of signals in channels will determine the behavior. The way to handle such a situation would be to manually decide the order in which transitions should be executed.

As the Cadvanced SDL to C Compiler is also intended to generate applications, process priority has been introduced as an additional feature. For more information about how to assign priorities to processes see sub-section "Assigning Priorities – Directive #PRIO" on page 2667.

The Ready Queue

The ready queue is a queue containing all process instances which have received a signal that can cause a transition, but which have not yet completed that transition. The ready queue is ordered firstly according to the priority and secondly according to insert time, that is a process which will be inserted last among the processes with the same priority, but before all processes with lower priority (high priority value = low priority). A process will never be inserted before the process currently executing, as pre-emptive scheduling is not used. In more detail:

- If a process outputs a signal to another process, which immediately can receive the signal, the receiving process will be inserted into the ready queue last among the processes with the same priority, but never before the currently executing process.
- If the processes currently executing a nextstate immediately can continue to execute another transition, it will be inserted into the ready queue last among the processes with the same priority. This means that it can remain as first process in the ready queue, but it can also be re-inserted somewhere else.
- If the receiving process at a timer output immediately can execute a transition as response to the received signal, the process will be inserted into the ready queue last among the processes with the same priority. This means that it can be inserted anywhere in the queue.

Enabling Conditions and Continuous Signals

Enabling conditions and continuous signals are additional concepts in SDL. The model for these concepts use repetitive signal sending, to have the expressions recalculated repeatedly. This model is not suitable during simulation, and definitely not acceptable in an application. We

have therefore used an implementation strategy closer to the described behavior of the concepts, rather than the model used to define the concepts.

Implementation Strategy

First we distinguish between those enabling conditions and continuous signals that are dynamic and those that are static, that is containing expressions that can or cannot change their value when the corresponding process is waiting in the state. The expression in a dynamic enabling condition or continuous signal contains some part that can change its value, even though the process does not execute any statements. Or, put more precisely, it contains at least one import, view, or reference to Now.

Static enabling conditions or continuous signals do not provide any problems or any execution overhead, except that the corresponding expressions have to be calculated at nextstate operations. Dynamic enabling conditions or continuous signals, however, have to repeatedly be recalculated. The strategy selected for these expressions is to recalculate them after each transition or timer output performed by any process (and additionally also before the monitor is entered within a transition). In other words, each process waiting in a state containing a dynamic enabling condition or continuous signal executes an implicit nextstate operation between each transition or timer output performed by other processes.

Synonyms

Synonyms

An SDL synonym is implemented either as a C macro (`#define`) or as a C variable. To be translated to a macro the expression defining the value of the synonym must be:

- Of one of the predefined SDL sorts (Integer, Real etc.).
- Possible to calculate at analyze time, i.e. it may only contain literals and operators defined in the predefined SDL sorts and other synonyms which are possible to calculate at analyze time.

All other synonyms are implemented as variables given their values at program start up.

The reason for raising this question is because it is relevant to the implementation of arrays and powersets. There are two different implementations for each of these concepts, see "Array" on page 2601 and "Powerset" on page 2602. An array in SDL can either be translated to an array in C or to a linked list in C. A powerset can either be translated to a bit array in C or to a linked list. The translation method is selected by looking at the index type. If the index type is a syntype with one limited range, the array and bit array scheme is used, otherwise the linked list is used.

If a synonym translated to a variable is used in a range condition of a syntype and the syntype is used as an index sort in an array or powerset instantiation, the linked list scheme is used to implement the array or powerset. The reason for this is that the length of the array cannot depend on a variable in C.

External Synonyms

External synonyms can be used to parameterize an SDL system and thereby also a generated program. The values that should be used for the external synonyms can either be read by the generated program during start up, or included as macro definitions into the generated code. The Cadvanced/Cbasic SDL to C Compiler can handle both these cases – it is not necessary to select which way should be used for each synonym until the program is compiled.

Using a Macro Definition

To use a macro definition in C to specify the value of an external synonym, perform the following steps:

1. Write the macro definitions on a file.

   Example 339: Macro Definition ––––––––––––––––––––––––––

   ```
   #define synonym1 value1
   #define synonym2 value2
   ```

   ––

   The synonym names are the SDL names (without any prefixes) and with any character not in letters, digits or underscore removed.

2. Introduce the following `#CODE` directive at the system level among the SDL definitions of, for example, synonyms, sorts, and signals but before any use of the synonyms.

Example 340: #CODE Directive

```
/*#CODE
#TYPE
#include "filename"
*/
```

If this structure is used, the value of an external synonym can be changed merely by changing the corresponding macro definition and recompiling the system.

Note:

When an application is created, macro definitions should be used for all external synonyms, as the function for reading synonym values stored on file is not available. (See below.)

Reading Values at Program Start up

The other way to supply the values of the external synonyms is to read the values at program start up. If there are any external synonyms that do not have a corresponding macro definition, it is possible to choose between supplying the values of the remaining external synonyms from the keyboard or to use a file containing the values.

When the application is started, the following prompt appears:

```
External synonym file :
```

- Press `<Return>` to indicate that the values should be read from the terminal.
- Or type the name of a file that contains the values and press `<Return>`.

If the user chooses to read the values from the terminal, he will be prompted for each value. In the other case the user should have created a file containing the external synonym names and their corresponding value according the following example:

Example 341: Values at Program Startup

```
synonym1 value1
synonym2 value2
```

The synonyms may be defined in any order.

Import – Export

These concepts are not implemented with the full semantics according to the model in the SDL recommendation. The model says that an imported value should be obtained using a signal interchange between the importer and exporter.

In the Cadvanced/Cbasic SDL to C Compiler we use a model where the imported value is directly obtained from the exporter, which of course makes the import operation much faster. However, the scheduling effect of the signal interchange is lost, as well as the change of SENDER in the involved processes. If these effects are important for an application, remote procedure calls can be used instead, see below.

Remote Procedure Calls

Remote procedure calls (RPC) have much in common with import/export, except that instead of obtaining one value, RPCs give the opportunity to execute a procedure in the exporting process. In the Cadvanced/Cbasic SDL to C Compiler, the model described in the SDL recommendation is used in detail to implement RPCs.

This means that a remote procedure call is translated to:

output of pCALL signal with all parameters.
nextstate in pWAIT, i.e. a implicit wait state.
input of pREPLY signal with all IN/OUT parameters.

In the exporting process there will be implicit transitions where the pCALL signal can be handled.

input pCALL.
call remote procedure with parameters from pCALL.
output pREPLY with the IN/OUT parameters.
nextstate -

For more details about this model, please see the SDL recommendation.

Procedure Calls and Operator Calls

In SDL-92 value returning procedures (and remote procedures) are introduced. This means that an SDL procedure can be called within an expression. In the Cadvanced/Cbasic SDL to C Compiler such procedure calls are implemented according to the model in the SDL recommendation, that is by inserting an extra CALL just before the statement con-

taining the value returning procedure call. The result from the call is stored in an anonymous variable, which is then used in the expression.

Example 342: Procedure Call ––––––––––––––––––––––––––––––––

```
TASK i := (call p(1)) + (call Q(i,k));
```

is translated to:

```
CALL p(1, Temp1);
CALL q(i, k , Temp2);
TASK i := Temp1 + Temp2;
```

––

> **Note:**
>
> The value returning procedure calls are transformed to ordinary calls, by adding a new IN/OUT parameter for the procedure result, last in the call.

Operators which are defined using operator diagrams, are according the models in the SDL recommendation, treated exactly as value returning procedure.

External Procedures And Operators

External procedures is a extension to SDL introduced in SDL-96. An external procedure is defined in a text symbol as a procedure heading:

```
procedure test; fpar a integer; returns integer;
external;
```

Instead of giving an implementation for the procedure the keyword external is inserted. The purpose of external procedures in SDL are to specify the existence of procedures without giving their implementation.

The Cadvanced/Cbasic SDL to C Compiler will generate no code for an external procedure declaration and will translate a call to such a procedure to an ordinary C function call. It is then up to the user to provide the C implementation of this function. Note that the code generator will in the generated function call use the name of the external procedure as it is. No prefix is inserted in this case, just as for external synonyms.

External operators are handled in the same way as external procedures. The name of the external operator is used in C just as it is. A quoted op-

erator will cause an infix operator to be generated, while operators with ordinary names will cause C function calls to be generated.

Any

There are two different applications of any. It is possible to write

```
any(SortName)
```

within an expression, or to write just

```
any
```

in a decision. The second case, with any in a decision, is implemented in the following way:

- Simulator:

  A question in the monitor giving the user a possibility to select the path to follow.

- Validator:

  The validator sees this as an non-deterministic choice and selects all possible paths.

- Applications:

  Should not be used!

The first case, the any(SortName) within an expression, is implemented using a random number generator to draw a random number of the given type.

> **Note:**
>
> any(Sort) where Sort is a syntype is only implemented if the syntype contains at most one range condition which is of the form a:b, that is one limited range. If it is a syntype of a real type, e.g. Real or Time, with a range condition it is not implemented.

If any(SortName) is used for a sort violating the note above, there will be a C compilation error on the symbol ANY_SortNameWithPrefix. This means that a user can implement any for such sorts himself by defining a C macro with this name, that implements any for the given sort. Such a macro should be inserted in the #TYPE section of a #ADT directive in the syntype.

Calculation of Receiver in Outputs

The Cadvanced/Cbasic SDL to C Compiler contains an algorithm that calculates the receiving process instance set, for outputs without TO, considering channels, signal routes, connection points, and via list. There are however a few restrictions for the algorithm:

- Outputs in process types, or in processes in block types or system types **cannot** be handled. The reason is that the same output might lead to different receivers in different instantiation.
- Paths (channels - signal routes) that lead into other units that are separate (see "Edit Separation" on page 136 in chapter 2, *The Organizer*) **cannot** be followed by the algorithm, as that would violate the separate generation scheme.
- Outputs in global procedures **cannot** be handled, as the receiver depends on the caller of the procedure.

This algorithm means that for an ordinary SDL-88 system, that is not generated using separate units, no information about the channels and signal routes are needed to direct signal to the correct receiver. For more information about the possible optimizations in applications, please see the compilation switch XOPTCHAN and the ADT PidLit ("The Data Type PIdLit" on page 3180 in chapter 63, *The ADT Library*). Please note that XOPTCHAN and PidLit is almost impossible to use if the SDL system contains system types, block types, or process types.

Abstract Data Types

This section is a reference to the abstract data types. The following topics will be discussed:

- We will have a look at the implementation of the predefined data types in SDL, see "SDL Predefined Types" on page 2588. We then discuss how user-defined abstract data types are translated, see "Translation of Sorts" on page 2595.
- Next implementation of operators and the possibility to include hand-coded C functions as implementation of the operators is presented, see "Implementation of User Defined Operators" on page 2609.
- Last, in "More about Abstract Data Types" on page 2634, we discuss more details about operators and the possibilities to include a hand-coded type definition in C to represent the SDL sort.

Removing un-used SDL Operators

When implementing an SDL system, you do not always use all available SDL operators. The Cbasic/Cadvanced SDL to C Compiler removes the declarations of unused operators, thus minimizing the code size of the generated application. Unused operators that are removed are:

- operators in predefined data types, for example substring, concatenate, length in the newtype Charstring, etc.
- operators defined in the predefined generators String, Array, Powerset, Bag
- special operators (and help functions) like assign, equal, default, make, extract, modify, free

The Cbasic/Cadvanced SDL to C Compiler performs the following steps to optimize the code:

1. Every C function that implements an operator is surrounded by an `#ifndef` definition.

Example 343 The `#ifndef` definition ––––––––––––––––––––––

```
#ifndef XNOUSE_AND_BIT_STRING
  /* function implementing the operator */
```

```
#endif
```

2. During the code generation, the usage of the operators in the translated SDL transitions is recorded.
3. The interdependencies between different operators are updated. For instance, an equal operator for a struct type may depend on equal operators for all its component types.
4. For each operator that is found to be unused, a `#define` definition is generated that removes the code for that operator. All the defines are placed in a file called *sdl_cfg.h.*

Example 344 The `#define` command

```
#define XNOUSE_AND_BIT_STRING
```

> **Note:**
> Even though the code size of the generated application is reduced, the code size of the generated C code is increased.

Manual override

In order to handle cases where operators are used invisibly from the Cbasic/Cadvanced SDL to C Compiler, for example in inline C code, you can manually override the automatic configuration of the unused operators.

In the code generation process, the Targeting Expert always generates a manual configuration file called `sct_mcf.h`. In this file you can list the unused operators that you have decided to include in the application. This is done by un-defining the previous definitions made in the `sdl_cfg.h` file.

The `sct_mcf.h` can be edited directly from Targeting Expert. Select **Edit Configuration Header File** from the **Edit** menu to open the file.

Example 345 The `#undef` command in the `sct_mcf.h` file

```
#ifdef XNOUSE_AND_BIT_STRING
#undef XNOUSE_AND_BIT_STRING
#endif
```

One section of the `sct_mcf.h` file, is dedicated for the manual edits. This section is marked with the text:

```
/* BEGIN User Code */
/*   END User Code */
```

The manual edits must be inserted between these to lines otherwise they will be deleted, as the `sct_mcf.h` file is re-generated each time you generate code. The information about the unused operators available in the `sdl_cfg.h` file is imported to the `sct_mcf.h` file. This allows you to quickly see which operators that are unused.

SDL Predefined Types

Mapping Table

Below is a table which summarizes the mapping rules between SDL and C, concerning the predefined types in SDL and their operators. Note that many of the operators are in C defined as macros, and expanded by the C preprocessor to simple operators in C.

SDL name/operator	C name/expression/operator
`Boolean`	`SDL_Boolean`
`False, True`	`SDL_False, SDL_True`
`not`	`xNot_SDL_Boolean`
`and`	`xAnd_SDL_Boolean`
`or`	`xOr_SDL_Boolean`
`xor`	`xXor_SDL_Boolean`
`=>`	`xImpl_SDL_Boolean`
`=, /=`	`yEqF_SDL_Boolean, yNEqF_SDL_Boolean`

SDL name/operator	C name/expression/operator
Character	SDL_Character
NUL SOH ...	SDL_NUL SDL_SOH ... (for all unprintable characters)
'a' 'b' ...	'a' 'b' ... (for all printable characters except ' and \)
'''', '\'	'\'', '\\'
chr	xChr_SDL_Character
num	xNum_SDL_Character
<, <=, >, >=	xLT_SDL_Character, xLE_SDL_Character, xGT_SDL_Character, xGE_SDL_Character
=, /=	xEqF_SDL_Character, xNEqF_SDL_Character
Charstring	SDL_Charstring
'aa'	SDL_CHARSTRING_LIT("Laa","aa")
mkstring	xMkString_SDL_Charstring
length	xLength_SDL_Charstring
first	xFirst_SDL_Charstring
last	xLast_SDL_Charstring
//	xConcat_SDL_Charstring
substring	xSubString_SDL_Charstring
=, /=	yEqF_SDL_Charstring, yNEqF_SDL_Charstring

SDL name/operator	C name/expression/operator
`Integer`	`SDL_Integer`
`0, 1` etc.	`SDL_INTEGER_LIT(0), SDL_INTEGER_LIT(1)` etc.
`+`	`xPlus_SDL_Integer`
`-` (monodic, dyadic)	`xMonMinus_SDL_Integer, xMinus_SDL_Integer`
`*`	`xMult_SDL_Integer`
`/`	`xDiv_SDL_Integer`
`mod`	`xMod_SDL_Integer`
`rem`	`xRem_SDL_Integer`
`float`	`xFloat_SDL_Integer`
`fix`	`xFix_SDL_Integer`
`<, <=, >, >=`	`xLT_SDL_Integer, xLE_SDL_Integer, xGT_SDL_Integer, xGE_SDL_Integer`
`=, /=`	`yEqF_SDL_Integer, yNEqF_SDL_Integer`
`Natural`	`SDL_Natural`
`Real`	`SDL_Real`
`12.45, ...`	`SDL_REAL_LIT(12.45, 12, 450000000)`
`-` (monodic, dyadic)	`xMonMinus_SDL_Real, xMinus_SDL_Real`
`+`	`xPlus_SDL_Real`
`*`	`xMult_SDL_Real`
`/`	`xDiv_SDL_Real`
`<, <=, >, >=`	`xLT_SDL_Real, xLE_SDL_Real, xGT_SDL_Real, xGE_SDL_Real`
`=, /=`	`yEqF_SDL_Real, yNEqF_SDL_Real`

SDL name/operator	C name/expression/operator
Pid	SDL_PId
Null	SDL_NULL
=, /=	yEqF_SDL_PId, yNEqF_SDL_PId
Duration	SDL_Duration
23.45	SDL_DURATION_LIT(23.45, 23, 450000000)
+	xPlus_SDL_Duration
- (monodic)	xMonMinus_SDL_Duration
- (dyadic)	xMinus_SDL_Duration
* (Duration * Real) * (Real * Duration)	xMult_SDL_Duration, xMultRD_SDL_Duration
/	xDiv_SDL_Duration
<, <=, >, >=	xLT_SDL_Duration, xLE_SDL_Duration, xGT_SDL_Duration, xGE_SDL_Duration
=, /=	yEqF_SDL_Duration, yNEqF_SDL_Duration
Time	SDL_Time
23.45	SDL_TIME_LIT(23.45, 23, 450000000)
+ (Time + Duration) + (Duration + Time)	xPlus_SDL_Time, xPlusDT_SDL_Time
- (result: Time)	xMinusT_SDL_Time
- (result: Duration)	xMinusD_SDL_Time
<, <=, >, >=	xLT_SDL_Time, xLE_SDL_Time, xGT_SDL_Time, xGE_SDL_Time
=, /=	yEqF_SDL_Time, yNEqF_SDL_Time
IA5String	SDL_IA5String
NumericString	SDL_NumericString
VisibleString	SDL_VisibleString
PrintableString	SDL_PrintableString

SDL name/operator	C name/expression/operator
Bit	SDL_Bit
not	xNot_SDL_Bit
and	xAnd_SDL_Bit
or	xOr_SDL_Bit
xor	xXor_SDL_Bit
=>	xImpl_SDL_Bit
=, /=	yEq_SDL_Bit, yNEq_SDL_Bit
Bit_string	SDL_Bit_String
not	xNot_SDL_Bit_String
and	xAnd_SDL_Bit_String
or	xOr_SDL_Bit_String
xor	xXor_SDL_Bit_String
=>	xImpl_SDL_Bit_String
mkstring	xMkString_SDL_Bit_String
length	xLength_SDL_Bit_String
first	xFirst_SDL_Bit_String
last	xLast_SDL_Bit_String
//	xConcat_SDL_Bit_String
substring	xSubString_SDL_Bit_String
bitstr	xBitStr_SDL_Bit_String
hexstr	xHexStr_SDL_Bit_String
=, /=	yEq_SDL_Bit_String, yNEq_SDL_Bit_String

SDL name/operator	C name/expression/operator
Octet	SDL_Octet
not	xNot_SDL_Octet
and	xAnd_SDL_Octet
or	xOr_SDL_Octet
xor	xXor_SDL_Octet
=>	xImpl_SDL_Octet
shiftl	xShiftL_SDL_Octet
shiftr	xShiftR_SDL_Octet
+	xPlus_SDL_Octet
-	xMinus_SDL_Octet
*	xMult_SDL_Octet
i2o	xI2O_SDL_Octet
o2i	xO2I_SDL_Octet
/	xDiv_SDL_Octet
mod	xMod_SDL_Octet
rem	xRem_SDL_Octet
bitstr	xBitStr_SDL_Octet
hexstr	xHexStr_SDL_Octet
<, <=, >, >=	yLT_SDL_Octet, yLE_SDL_Octet, yGT_SDL_Octet, yGE_SDL_Octet
=, /=	yEq_SDL_Octet, yNEq_SDL_Octet

SDL name/operator	C name/expression/operator
Octet_string	SDL_Octet_String
mkstring	xMkString_SDL_Octet_String
length	xLength_SDL_Octet_String
first	xFirst_SDL_Octet_String
last	xLast_SDL_Octet_String
//	xConcat_SDL_Octet_String
substring	xSubString_SDL_Octet_String
bitstr	xBitStr_SDL_Octet_String
hexstr	xHexStr_SDL_Octet_String
bit_string	xBit_String_SDL_Octet_String
hex_string	xHex_String_SDL_Octet_String
=, /=	yEq_SDL_Octet_String, yNEq_SDL_Octet_String
Object_identifier	SDL_Object_Identifier
mkstring	xMkString_SDL_Object_Identifier
length	xLength_SDL_Object_Identifier
first	xFirst_SDL_Object_Identifier
last	xLast_SDL_Object_Identifier
//	xConcat_SDL_Object_Identifier
substring	xSubString_SDL_Object_Identifier
=, /=	yEq_SDL_Object_Identifier, yNEq_SDL_Object_Identifier
NULL (sort)	SDL_Null
NULL (literal)	SDL_NullValue
=, /=	yEq_SDL_Null, yNEq_SDL_Null

C Definitions

We will here discuss the types and macros supplied by the runtime library in the Cadvanced/Cbasic SDL to C Compiler for the predefined types in SDL. These macros and extern definitions for functions can be found in the file `sctpred.h`, except for the Pid sort which is handled in the file `scttypes.h`.

Note:

For more information about the Charstring sort, see the section "Handling of the Charstring Sort" on page 2618.

Translation of Sorts

The following data types are handled by the Cadvanced/Cbasic SDL to C Compiler:

- *Predefined Types*
- *Enumeration Type*
- *Struct*
- *Choice*
- *Array*
- *String*
- *Powerset*
- *Bag*
- *Ref, Own, Oref*
- *Syntypes*
- *Inheritance*

Predefined Types

All the predefined data types (Integer, Natural, Boolean, Character, Charstring, Real, Time, Duration, Pid, Bit, Bit_string, Octet, Octet_string, Object_identifier, IA5String, NumericString, PrintableString, and VisibleString) are completely handled. The name of these types in the generated C code will be SDL_Integer, SDL_Natural, SDL_Boolean, and so on. The translation rules for these types and their operators are discussed in more detail in the "SDL Predefined Types" on page 2588.

Enumeration Type

A sort which is not a struct and does not contain any inheritance or generator instantiation, but which contains a literal list, is seen as an enumeration type. See the example below. Such a type is translated to **int**, together with a list of defines where the literals are defined as 0, 1, 2, and so on. As in all examples in this sub-section, the prefixes, which are added to names when they are translated to C, are not shown. The prefixes are added to make sure that no name conflicts occur in the generated program. For more information about prefixes see "Names and Prefixes in Generated Code" on page 2663.

Example 346: Enumeration Type

```
NEWTYPE EnumType
  LITERALS Lit1, Lit2, Lit3;
ENDNEWTYPE;
```

is translated to:

```
typedef XENUM_TYPE EnumType;
#define Lit1 0
#define Lit2 1
#define Lit3 2
```

Where the macro `XENUM_TYPE` is defined in sctpred.c as:

```
#ifndef XENUM_TYPE
#define XENUM_TYPE int
#endif
```

This means that all enum types will be int types, except if the macro `XENUM_TYPE` is redefined by the user (to `unsigned char` for example). An enum type with 256 or more values will always be of type int and will not be affected by the macro `XENUM_TYPE`.

Struct

An SDL struct is translated to a struct in C, as can be seen in the example below.

Example 347: Struct

```
NEWTYPE Str STRUCT
  a Integer;
  b Boolean;
  c Real;
ENDNEWTYPE;
```

is translated to:

```
typedef struct {
  SDL_Integer a;
  SDL_Boolean b;
  SDL_Real c;
} Str;
```

All the properties of a struct in SDL are preserved in the C code.

The predefined operators extract! and modify! are implemented as component selections in the struct in the same way as in SDL, that is, if S is a variable of type Str, then S!a in SDL is translated to S.a in C.

The predefined operator make!, which is a constructor of a struct value, is implemented by generating a Make function in C. This means that the expression "(. 12, true, 0.22 .)" in SDL is in principle translated to the C function call Make(12, true, 0.22).

The components of a struct may be of any sort that the code generator can handle. A component may, however, not directly or indirectly refer to the struct sort itself. As an example the sort Str above may not have a component of sort Str. In such a case the translation to a C struct would not any longer be valid.

There are some extensions to SDL that are handled by the code generator. It is possible to define bit fields, i.e, to define the size of components (as in C) and to have optional components and components with initial values (as in ASN.1). Examples are shown below.

Example 348: Struct with bit fields

```
NEWTYPE str STRUCT
  a Integer : 4;
  b Integer : 4;
            : 0;
  c UnsignedInt : 2;
  d Integer;
ENDNEWTYPE;
```

is translated to:

```
typedef struct str_s {
    SDL_Integer  a : 4;
    SDL_Integer  b : 4;
    int : 0;
    UnsignedInt  c : 2;
    SDL_Integer  d;
```

```
} str;
```

Note that only Integer and UnsignedInt should be used in bit field components.

Example 349: Struct

```
NEWTYPE str STRUCT
  a, b integer;
  c    Boolean OPTIONAL;
  d    str2 OPTIONAL;
  e    Charstring := 'telelogic';
  f    arr3 := (. 11 .);
ENDNEWTYPE;
```

is translated to:

```
typedef struct str_s {
    SDL_Integer  a;
    SDL_Integer  b;
    SDL_Boolean  c;
    SDL_Boolean  cPresent;
    str2  d;
    SDL_Boolean  dPresent;
    SDL_Charstring  e;
    SDL_Boolean  ePresent;
    arr3  f;
    SDL_Boolean  fPresent;
} str;
```

Both optional components and components with initial values have a Present flag. This is according to ASN.1 and the translation of ASN.1 to SDL defined in Z.105. The present flag for a component with initial value is true if the component contains its default value otherwise false (the Present flag is used to determine code for some ASN.1 encoding scheme). The present flag for an optional component is false until the component is assigned a value. In SDL the present flags can only be accessed through operators and cannot be changed.

Union

Please see also the CHOICE concept presented below, as it usually provides a better and more secure solution to the same kind of problems.

Using the directive #UNION (see example below) it is possible to tell the Cadvanced/Cbasic SDL to C Compiler to generate a union according to the following example:

Example 350: Union ––

```
NEWTYPE Str /*#UNION*/ STRUCT
  tag integer;
  a integer;
  b Boolean;
  c real;
ENDNEWTYPE;
```

is translated to:

```
typedef struct {
  SDL_Integer tag;
  union {
    SDL_Integer a;
    SDL_Boolean b;
    SDL_Real c;
  } U;
} Str;
```

––

The first component in the struct is assumed to be a tag value indicating which of the union components that are active. The tag should either be integer or an enumeration type. Tag value 0 or first enumeration literal is used to indicate that the first of the remaining components are active, and so on. On the SDL level a #UNION struct should be handled just like any other struct. It is up to the code generator to generate the correct code for operations on the struct, like assignment, test for equality, component selection, and so on.

> **Note:**
>
> It is completely up to the user to make certain that only valid components in a #UNION struct are accessed. During simulation, however, tests are inserted to ensure that only valid components are accessed.

UnionC

By using the directive #UNIONC according to the example below, it is possible to tell the Cadvanced/Cbasic SDL to C Compiler to generate a true C union.

Example 351: UnionC ––––––––––––––––––––––––––––––––––––––

```
NEWTYPE Str /*#UNIONC*/ STRUCT
  a integer;
  b Boolean;
  c real;
ENDNEWTYPE;
```

is translated to:

```
typedef union {
  SDL_Integer a;
  SDL_Boolean b;
  SDL_Real c;
} Str;
```

Note:

The #UNIONC directive is not recommended for use as the Cadvanced/Cbasic SDL to C Compiler cannot give any support for checking the validity of component selection. Both the #UNION directive and the CHOICE concept discussed below are much better.

Note also that pointer types, including Charstrings are not allowed in #UNIONC structs, as it is not possible to know when to allocate and deallocate memory for such components.

Choice

Choice, which is an SDL extension originating from the needs when translating ASN.1 to SDL and which is included in SDL-2000, can be used to express a union with implicit tag.

Example 352: Choice ----------

```
NEWTYPE Str CHOICE
  a integer;
  b Boolean;
  c real;
ENDNEWTYPE;
```

is translated to:

```
typedef enum {a, b, c} StrPresent;
typedef struct {
  StrPresent Present;
  union {
    SDL_Integer a;
    SDL_Boolean b;
    SDL_Real c;
  } U;
} Str;
```

The component Present, which is the tag field, and its type (`StrPresent` in the example above) are both available in SDL. The Present component can in SDL be accessed, but not changed, through:

- component selection, i.e. by `Variable!Present`, i.e. it is possible to for example test: `V!Present = a`
- the operators aPresent, bPresent, or cPresent, which returns true or false depending on if the component is active or not.

The Present component is automatically set by the code generator when a component in the choice is given a value.

Note that during simulations and validations, it is automatically tested that a component “is present” when an attempt is made to access the component. A run-time error is issued if this is not the case.

Array

Instantiations of the predefined generator array can be handled by the code generator with the following restriction: The component and index sort may be any sorts that the code generator can handle, but may not directly or indirectly refer to the array type itself (see also the previous paragraph on struct).

If the index sort is a discrete sort, with one closed interval of values, that is of the following sorts:

- Character
- Boolean
- Octet
- Bit
- A sort that is considered as an enumeration type
- Syntypes of integer, character, Boolean, Octet, Bit, and enumeration types. The subtypes may only have one range condition that specifies a closed interval of values,

then the SDL array is translated to a struct containing an element which is an array in C.

If the index sort is **not** one of the sort in the enumeration above, the SDL array is translated to a linked list. The list head contains the default value for all possible indexes, while the list elements contain value pairs, (index_value, component_value), for each index having a component value not equal to the default value.

Example 353: Array ––

```
SYNTYPE Syn = integer
  CONSTANTS 0:10
```

```
ENDSYNTYPE;

NEWTYPE Arr ARRAY(Syn, real)
ENDNEWTYPE;
```

is translated to:

```
typedef SDL_Integer Syn;
typedef struct {
 SDL_Real A[11];
} Arr;
```

All the properties of an array in SDL are preserved in the C code.

The predefined operators extract! and modify! are implemented as component selection of the array in C in the same way as in SDL, so if AVar is a variable of type Arr, and Index is a valid index expression, then `AVar(Index)` in SDL is translated to `AVar.A[Index]` in C. In the case of a link list implementation of the array, component selection is made through function calls.

The predefined operator make!, which is a constructor of an array value, is implemented by generic Make function in C.

String

Instantiations of the predefined generator string can be handled by the code generator with the following restriction: The component sort may be any sorts that the code generator can handle, but may not directly or indirectly refer to the string type itself.

There are two translation schemes for Strings. The directive #STRING decide whether the string should be translated to linked list or to an array. For the #STRING directive please see "Alternative Implementations of the String Generator – Directive #STRING" on page 2674.

Strings are translated to linked list containing one element for each element in the string value. Operations and component selection in string sorts are fully supported.

Powerset

Instantiations of the predefined generator powerset can be handled by the code generator with the following restriction: The component sort may be any sorts that the code generator can handle, but may not directly or indirectly refer to the powerset type itself.

There are two translation schemes for powersets. If the component sort fulfills the conditions for index sorts mentioned in the subsection about arrays above ("Array" on page 2601), an array of 32-bit integers are used. Each bit will be used to represent a certain element whether it is a member of the powerset or not. If this is not the case, a linked list of all elements that are member of the set, is used to represent the powerset. All the available operations defined for Powersets in SDL are supported.

Bag

The Bag generator, which is introduced in SDL in Z.105, i.e. in the mapping from ASN.1 to SDL, is similar to powerset. However, it is possible to have several elements with the same value in a bag. A bag is always translated into a linked list, with one element for each value that is a member of the bag. Each element contains the value and the number of occurrences of this value.

Ref, Own, Oref

These generators represent pointers with different properties. They are all translated to pointers in C.

Syntypes

Syntypes may be defined for any sort that the code generator can handle, giving a new name for the sort and possibly a new default value for variables of the sort. Range conditions that restrict the allowed range of values are also allowed.

A syntype is translated to a type equal to the parent type using typedef. The check that a variable of a syntype is only assigned legal values is implemented in a test function that is generated together with the type definition. An attempt to assign an illegal value to such a variable will be reported as an SDL dynamic error. If the syntype is can be used as index sort in an array and the generated type in C would become an array, there will also be a test function that can be used to check that an index value is within its range in an array component selection.

Example 354: Syntypes ––––––––––––––––––––––––––––––––––––

```
SYNTYPE Syn = integer
  CONSTANTS 0:10
ENDSYNTYPE;

SYNTYPE Syn2 = integer
```

```
  CONSTANTS <0, =2, >=10
ENDSYNTYPE;

SYNTYPE Arr1 = Arr
  DEFAULT (. 2.0 .);
ENDSYNTYPE;
/* Arr defined above */
```

is translated to:

```
typedef SDL_Integer Syn;
typedef SDL_Integer Syn2;
typedef Arr Arr1;
```

Inheritance

A type that inherits another type is translated to a type equal to the parent type using a typedef.

Default Values

Default values are fully supported for all sorts that the code generator can handle, both if a default value is given in a sort definition and if an initial value is given in a variable definition (DCL).

Default values will also be assigned to all variables and components which do not have a default value specified in SDL. The reason for this is to avoid handling undefined variables in C, which might give serious problems and unexpected behavior of an executing program. The values selected by the code generator in such a case can be found below.

> **Note:**
>
> This is a deviation from SDL-92. It means that the generated program does not handle the value undefined for any type.

If no default value is given in the sort and no start value is given in the data definition (DCL) for a variable, the variable will be set to 0 by using a memset to 0.

Operators

In SDL-92, it is possible to define the behavior of operators in ADTs directly in SDL, using operator diagrams or operator implementations in SDL textual form. Such operators are translated to C by the Cadvanced/Cbasic SDL to C Compiler, and none of what is said below is

valid for such an operator. It is also possible to specify that an operator is external. In this case the code generator assumes that a C function with the name used in SDL exists and translates calls to the external operator directly to calls to the C function.

A user defined operator in an SDL sort definition, which is not defined by an operator diagram, is translated to a C function which asks the user for the result of the operation. At a call of an operator, the user is supplied with information describing what the operator and the sort are called, and given information about the parameter values. You are then requested to answer with the result value. If you press <Return> at the prompt for the result, the default value of the actual result type is returned. If the operator does not have a result type, no question is asked.

Example 355: Operator

```
Operator Op in sort S is called.
Parameter 1: true
Parameter 2: 10
Enter value (integer) : 12
```

assuming that newtype S contains an operator

```
Op: Boolean, Integer -> Integer;
```

More about operator implementation, both parameter passing and how to include implementations written in C can be found in the next two sections.

Literals

In sorts that are translated to enumeration types in C, literals are obviously handled by the code generator. In sorts that are not enumeration types, literals are treated as operators without parameters and are handled in exactly the same way as user defined operators.

> **Note:**
> The Cadvanced/Cbasic SDL to C Compiler does not permit naming of literals using name class literals or character strings.

Axioms and Literal Mappings

Axioms and literal mapping are allowed by the code generator in sorts, but are completely ignored.

Parameter Passing to Operators

For performance reasons the data types in SDL have been divide in two groups, simple, small types that are passed as values and structured, larger types that are passed as references (addresses).

Types passed as addresses (structured types)
Bit_string
Octet_string
Object_identifier
Struct types (including #UNION, #UNIONC)
Choice types
Instantiations of generator Powerset
Instantiations of generator Bag
Instantiations of generator Array
Instantiations of generator String
Instantiations of generator Carray
Syntypes of a type in this list
Types that inherit a type in this list

Table 1: Types Passed as Addresses

Types passed as values (simple types)
Integer
Real
Natural
Boolean
Character
Time
Duration
PId
Charstring
Bit
Octet
IA5String
NumericString
PrintableString
VisibleString
NULL
Enumeration types
Instantiations of generator Ref, Own, ORef
Syntypes of a type in this list
Types that inherit a type in this list

Table 2: Types Passed as Values

Note:

For types represented as pointers (Charstring, including its syntypes, Ref, Own, ORef), the pointers, not the addresses of the pointers, are passed as parameters.

The parameter passing for operators implemented in C works as follows (for Cmicro the mechanism described below is also used for operator diagrams and procedures):

In parameters:

- Passed as a value in C if the type is in the list “Passed as value”. This means that the parameter type in C is the same type as in SDL.
- Passed as an address in C if the type is in the list “Passed as address”. This means that the C parameter is (SDL_type *) if the type in SDL is SDL_type.

In/Out parameters:

Parameters are always passed as addresses, i.e the C parameter is (SDL_type *) if the type in SDL is SDL_type.

Operator result:

- If the result type is in the list “Passed as value”, the C function result type will be the same as in SDL.
- If the result type is in the list “Passed as address”, two things are changed. Firstly, the C result type will be (SDL_type *), i.e the result will be an address. Secondly, an extra parameter is inserted last in the C function. This parameter is also of type (SDL_type *) and is used as a location to store the result of the function. At an operator call, a “dummy” variable should be passed as the actual parameter. The C function can then use this to store the result of the operator and should return the variable again as result.

Example 356: ––

Assume that struct1 is a newtype struct in SDL.

```
operators
 X : integer, in/out integer -> integer;
 Y : struct1, in/out struct1 -> struct1;
```

The C prototypes for these operators are:

```
SDL_Integer X (SDL_Integer, SDL_Integer *);
struct1 * Y (struct1 *, struct1 *, struct1 *);
```

The example implementations are:

```
SDL_Integer X
  (SDL_Integer Param1, SDL_Integer *Param2)
```

```
{
  *Param2 = *Param2+Param1;
  return *Param2;
}

struct1 * Y (struct1 *Param1,
             struct1 *Param2,
             struct1 *Result)
{
  /* implementation assuming struct1 to contain
     two integers */
  (*Param2).comp1 = (*Param2).comp1+(*Param1).comp1;
  (*Param2).comp2 = (*Param2).comp2+(*Param1).comp2;
  *Result = *Param2;
  return Result;
    /* always return the last, extra, parameter */
}
```

Note: VERY IMPORTANT

As IN parameters are passed as addresses for structured types, changing such a parameter inside the operator might have undesired effects. A variable passed as actual parameter is then also changed. If you want to change the formal parameter copy it first to a operator local variable.

For Cadvanced/Cbasic this rule applies to operators implemented in C. For Cmicro this rule also applies to operators and procedures defined in SDL.

Implementation of User Defined Operators

Including Implementations of Operators

In a previous subsection, the default behavior of the Cadvanced/Cbasic SDL to C Compiler concerning operators (not defined in operator diagrams) and literals were described. If you do not specify otherwise interactive functions are generated, which, in each case, will ask you for the operator result or literal value. This is a fast way of getting started, but you will probably find it tedious in the long run, especially if you are using abstract data types extensively. To cope with this problem and to make it possible to generate applications, the code generator offers a possibility to include implementations written in C of the operator and

literal functions. This possibility can be used as an alternative to operator diagrams or operators defined using SDL textual form, where the operator is defined directly in SDL.

When the choice between an implementation in SDL or in C is to be made there are a few things to consider:

- There is always problems when mixing languages, for example how are C names for SDL entities constructed.
- Checking of SDL is performed by the SDL Analyzer, which will find problems much earlier than the C compiler checking C code. Also pointing to the error will be more accurate in SDL.
- SDL implementations will be more portable and might benefit from future improvements in the SDL Compiler.
- The risk for backward compatibility problems in future releases of the SDL suite will be less for an SDL implementation.
- However, a C implementation might be more efficient or you might already have a corresponding C function.

So it is not obvious if SDL or C implementations should be used. However, we recommend SDL if there are no specific reasons for using C. Note also that the SDL extension described in "Grammar for the Algorithmic Extensions" on page 146 in chapter 3, *Using SDL Extensions, in the SDL Suite Methodology Guidelines* could be very useful when writing implementations in SDL.

It is possible to choose between two alternatives to implement operators and literal functions:

- Q (question)
  This is the default value and specifies that the code generator should generate the interactive routines describe above.
- B (body)
  This specifies that the code generator should generate the heading of the operator and literal functions, while the user must supply the bodies of the functions.

Note:

The C functions are divided into a function heading (extern or static declaration) and a function body.

An example of a function heading (extern declaration) is:

Example 357: Implementing an Operator

```
extern SDL_Integer Max
  (SDL_Integer Para1,
   SDL_Integer Para2);
```

while the corresponding function body is:

```
SDL_Integer Max
  (SDL_Integer Para1,
   SDL_Integer Para2)
{
 if (Para1 > Para2)
   return Para1;
 return Para2;
}
```

The main reason for this division of functions into heading and body is the separate compilation scheme used in C. If, for example, an abstract data type is defined in a system and used in a process in the system, and the process is generated on a separate file, then there has to be a module interface file (a `.h` file) for the system containing the external interface (types, extern declarations of functions and so on). The interface file should then be included in the file generated for the process.

Even if separate compilation is not used, the division of functions into heading and body is useful. By having static declarations of the functions, the order in which functions must be defined is relaxed. If static declarations were not used, a function could only call the functions that are defined textually before the actual function.

To select the way the Cadvanced/Cbasic SDL to C Compiler should generate code for operators and literals, code generator directives are used. A code generator directive is an SDL comment with the first characters equal to '#', followed by a sequence of letters identifying the directive. In this case the letters are ADT (for Abstract Data Type) and OP (for operator). An ADT directive and a OP directive should thus look like:

```
/*#ADT */ /*#OP */
```

The text is not case sensitive.

OP directives are recognized at two different positions in an abstract data type:

- Directly after the name of a literal
- Directly after the semicolon ending the definition of an operator.

ADT directives are recognized immediately before the reserved word ENDNEWTYPE (or ENDSYNTYPE).

Example 358: Implementing an Operator (#ADT) ––––––––––––––––––

```
NEWTYPE Str STRUCT
  a integer;
  b Boolean;
  c real;
ADDING
LITERALS
  Lit1 /*#OP */,
  Lit2 /*#OP */;
OPERATORS
  Op1 :Str,integer -> Str; /*#OP */
  Op2 :Str,Boolean -> Str; /*#OP */
/*#ADT */
ENDNEWTYPE;
```

––

At each of the positions after a literal name or operator definition, there is a possibility to specify how this literal or operator should be implemented. In the directive immediately before ENDNEWTYPE the default implementation technique can be given. When the code generator determines how to generate code for a literal or an operator, it first looks for an OP directive after the literal name or operator definition. If no such directive is found it looks for a directive immediately before ENDNEWTYPE. If no ADT directive is found here, the generation technique Q (question) is assumed. For Cmicro, Q is not used, so the default is B.

An OP or ADT directive specifying a generation technique should have the following structure:

```
/*#OP (B) */ /*#ADT (B) */
```

The letter between the parentheses should be either Q (question) or B (body). The interpretation of Q and B was explained earlier. If B has been specified for any operators or literals, then the C code for these functions must be supplied by the user. This code should be placed in the #BODY section in the ADT directive, according to the following example:

Example 359: Implementing an Operator (#ADT) ––––––––––––––––––

```
/*#ADT (B)
#BODY
C code, representing bodies of functions
```

```
*/
```

The section name, i.e #BODY, must be given on a line of its own and must have the # character in the first position of the line. Upper case and lower case letters are as usual considered to be equal. If the section is empty, the section name can also be removed.

Note:

The Cadvanced/Cbasic SDL to C Compiler will **not check the consistency** between the specification of implementation techniques and the actual code included in the body section. This check is, together with checking the C code for syntactic and semantic errors, **left to the C compiler**.

Unfortunately it is not possible to have C comments within the code that is included in a #ADT directive, as SDL and C use the same symbols for start and end of comments. If a C comment is included, the SDL Analyzer will consider the end of the C comment as the end of the SDL comment. Instead a C macro called COMMENT can be used according to the examples below. Note that there might be some compiler dependent restriction of the character set allowed within the COMMENT macro. For example, the character ';' might not be allowed.

Example 360: Comment in ADT

```
COMMENT(This is a comment)
COMMENT(These comments may not contain commas \
        and should have a backslash at each \
        line break)
COMMENT((By having double parenthesis, any text
         can be entered into the comments. Some
         compilers might not allow everything.))
```

The function headings representing literals and operators are determined by their corresponding definition in SDL. The number of parameters, their types, the result type of the function and function name are all defined in SDL. In the example above, where the struct Str is defined, there are two literals (Lit1 and Lit2) and two operators (Op1: Str, integer –> Str; and Op2: Str, Boolean –> Str;). The type Str will be passed as an address, so the parameter passing rules described previously have to be applied. The function heading of the corresponding C functions should be:

Example 361: Implementing an Operator ––––––––––––––––––––––

```
extern Str* Lit1 (Str*);
extern Str* Lit2 (Str*);
extern Str* Op1 (Str*, SDL_Integer, Str*);
extern Str* Op2 (Str*, SDL_Boolean, Str*);
```

––

The function bodies, which should be supplied by the user if B is specified in the OP or ADT directive, are ordinary C functions.

Example 362: Implementing an Operator ––––––––––––––––––––––

```
Str* Lit1 (Str* Result)
{
 Result->a = 2;
 Result->b = false;
 Result->c = 10.0;
 return Result;
}

Str* Op1 (Str* P1, SDL_Integer P2, Str* Result),
{
  *Result = *P1;
  Result->a = P1->a + P2;
  return Result;
}
```

––

Before it is possible to give a complete example of an abstract data type with implementation of its operators supplied as C functions, it is necessary to look at the problem of names. When a name of some object in SDL is translated to C, a suitable sequence of characters, a prefix, is added to the SDL name, to make the name unique in the C program, see also "Names and Prefixes in Generated Code" on page 2663. This strategy is selected in the Cadvanced/Cbasic SDL to C Compiler to avoid name conflicts in the generated code, but it makes it also impossible to predict the full name of, for example, a type or a function, in the generated program. To handle this problem the user can tell the code generator to translate a name in the C code in the same way as SDL names are otherwise translated. This is specified by enclosing the SDL name between '#(' and ')' in the C code. The two functions in the previous example and their headings would then become:

Example 363: Including SDL name in C Code ––––––––––––––––––

```
extern #(Str)* #(Lit1) (#(Str)*);
extern #(Str)* #(Op1) (#(Str)*, SDL_Integer,
                       #(Str)*);
```

```
#(Str)* #(Lit1) (#(Str)* Result)
{
 Result->a = 2;
 Result->b = false;
 Result->c = 10.0;
 return Result;
}

#(Str)* Op1 (#(Str)* P1, SDL_Integer P2,
             #(Str)* Result),
{
  *Result = *P1;
  Result->a = P1->a + P2;
  return Result;
}
```

This facility to access an SDL name in C code is described in more detail in the section "Accessing SDL Names in C Code – Directive #SDL" on page 2654. A few observations concerning the example above might be appropriate:

1. The predefined sorts in SDL, that is for example integer, natural, Boolean have the names SDL_Integer, SDL_Natural, SDL_Boolean, and so on in the generated code. These types should not be enclosed between '#(' and ')'.

2. The component names of a struct are unchanged in the struct implementation in C, which means that struct components should not be enclosed between '#(' and ')' either.

Two Examples of ADTs

We now give two complete examples of abstract data types.

Example 364: ADT Example----------

```
NEWTYPE Str STRUCT
  a Integer;
  b Boolean;
  c Real;
  ADDING LITERALS
    Lit1;
  OPERATORS
    Op1 : Str, Integer -> Str;
    Op2 : Str, Boolean -> Str;
/*#ADT (B)
#BODY
#(Str)* #(Lit1) (#(Str)* Result)
```

```
{
  Result->a = 2;
  Result->b = SDL_False;
  Result->c = 10.0;
  return Result;
}

#(Str)* #(Op1) (#(Str)* P1, SDL_Integer P2,
                #(Str)* Result)
{
  *Result = *P1;
  Result->a = P1->a + P2;
  return Result;
}

#(Str)* #(Op2) (#(Str)* P1, SDL_Boolean P2,
                #(Str)* Result)
{
  if (P2)
    *Result = *P1;
  else
    (void)#(Lit1)(Result);
  return Result;
}
*/
ENDNEWTYPE;
```

The example above should be compared with the same example written in SDL. Note that the literal in the previous example is replaced with an operator without parameters. The algorithmic extensions described in "Grammar for the Algorithmic Extensions" on page 146 in chapter 3, *Using SDL Extensions, in the SDL Suite Methodology Guidelines* is also used as they provide a powerful way to write textual algorithms.

Example 365: ADT Example in pure SDL ----------

```
NEWTYPE Str STRUCT
  a Integer;
  b Boolean;
  c Real;
  OPERATORS
    Lit1 : -> Str;
    Op1 : Str, Integer -> Str;
    Op2 : Str, Boolean -> Str;
OPERATOR Lit1 RETURNS Str
{
   RETURN (. 2, false, 10.0 .);
}
OPERATOR Op1 FPAR P1 Str, P2 Integer RETURNS Str
{
   DCL Result Str;
   Result := P1;
```

```
    Result!a := P1!a + P2;
    RETURN Result;
}
OPERATOR Op2 FPAR P1 Str, P2 Boolean RETURNS Str
{
  IF (P2)
    RETURN P1;
  RETURN Lit1;
}
ENDNEWTYPE;
```

Example 366: ADT Example

```
SYNTYPE Index = Integer CONSTANTS 1:10
ENDSYNTYPE,

NEWTYPE A Array(Index, Integer)
  ADDING LITERALS
    Zero /*#OP (B) */;
  OPERATORS
    Add : A, A -> A; /*#OP (B) */
    Sum : A -> Integer;
/*#ADT()
#BODY
#(A)* #(Zero) (#(A)* Result)
{
  SDL_Integer i = 0;
  GenericMakeArray(Result,
    (tSDLTypeInfo *)&ySDL_#(A), &i);
  return Result;
}

#(A)* #(Add) (#(A)* P1, #(A)* P2, #(A)* Result)
{
  int I;
  for (I = 1; I<=10; I++)
    Result->A[I] = P1->A[I] + P2->A[I];
  return Result;
}
*/
ENDNEWTYPE;
```

Note that no body is supplied for the operator Sum as the default implementation strategy for operators, which should be used for Sum, is Q (question). The `GenericMakeArray` function used to implement the literal is a generic function that constructs array values. The details for this function will be described later in this section.

For more information about the functions and types (supplied by the runtime library in the Cadvanced/Cbasic SDL to C Compiler and con-

tained in generated code) that can be useful when implementing operators in C, see "SDL Predefined Types" on page 2588, and last in "More about Abstract Data Types" on page 2634.

Error Situations in Operators

In the C function used to implement operators (and literals) it is possible to define error situations and handle them as ordinary SDL run-time errors. The C library function xSDLOpError, with the following prototype:

```
extern void xSDLOpError(
  char *OpName,
  char *ErrText )
```

can be used for this purpose.

Example 367: Error Handler in Operator

Example of use:

```
  if ( strlen(C) <= 1 ) {
#ifdef XECSOP
    xSDLOpError("First in sort Charstring",
                "Charstring length is zero." );
#endif
    return SDL_NUL;
  } else
    return C[1];
```

This is a simplified version of the test in the function for the operator First in the sort Charstring. Here the error situation is when we try to access the first character in a charstring of length 0. In this case the xSDLOpError is called and a default value is returned (NUL). By including the xSDLOpError call between `#ifdef XECSOP - #endif` the function is only called to report the error if error checks are turned on. The first parameter to xSDLOpError should identify the operator and the sort, while the second parameter should describe the error.

Handling of the Charstring Sort

The SDL sort Charstring is implemented as char * in C.

Note:

This means that the value NUL (ASCII character 0) **cannot** be part of a Charstring, as this value is used as string terminator in C (this is checked by the library functions for Charstring).

The code generator and the library functions for the Charstring operators use the first character (index 0) in the C string to indicate the status of the string. If the first character is:

- 'V'
  the string is assigned to an SDL variable and may not be changed in any way.
- 'L'
  the string is a C char * literal, and may of course not be changed.
- 'T'
  the string is a temporary result from a function returning a Charstring. This memory should either be assigned to an SDL variable or returned to the pool of free memory.

All the library functions for Charstrings handle memory in an appropriate way. A user only has to take the extra character in to account, when Charstrings are handled in C. Any Charstring function parameters having a 'T' as first character must be handled according to the discussion above. A function that returns a Charstring and that creates new temporary memory to store the result, should assign the value 'T' to the first character in the Charstring.

As pointers and dynamic memory are used to implement Charstrings, it is necessary to be careful when Charstrings are handled in C code, which we show in two examples.

Example 368: Equal Test on Charstring Sort

If the `C` operator `==` is used to check if two charstrings are equal, then the actual test that is performed is to see if the two pointer values to the data areas representing the characters in the string are equal.

To check if the characters in the charstrings are equal the equal function should be used:

```
yEqF_SDL_Charstring
```

Example 369: Assignment on Charstring Sort

If the C assignment operator, =, is used to assign the value of one charstring variable (C1) to another charstring variable (C2), then two things will go wrong:

1. The memory used to represent the old value of C1 is lost and can never be reused.
2. C1 and C2 now refer to the same memory area, which means that if one of the variables is changed the other will also be changed. This leads to unpredictable behavior of the program.

The correct way to handle assignment of charstrings is to use the routine:

```
yAssF_SDL_Charstring
```

The problems mentioned above can of course also occur if a struct or array containing charstring components (or subcomponents) is handled carelessly. It is, for example, necessary to use the generic equal and assign functions to perform equal test and assignment.

To avoid problems one should be aware that Charstring is implemented as char * in C and take the consequences thereof. There are a number of help functions (that implement the operators for the Charstring sort) supplied in the runtime library that might be helpful when handling Charstrings. See "SDL Predefined Types" on page 2588).

Other Types Containing Pointers

The principal discussion about Charstrings in the previous section is also relevant for all other types containing pointers. Such types are:

- Bit_string
- Octet_string
- Object_identifier
- Strings (not #STRING)
- General Arrays
- General Powersets
- Bags

All these types contain a boolean component, IsAssigned, that gives the status of the data area. IsAssigned serves the same purpose as the first extra character in a Charstring and has to be treated in a similar way.

- IsAssigned equal to false means that this data area is a temporary result from a function returning the data type. This memory should either be assigned to an SDL variable or returned to the pool of free memory.
- IsAssigned equal to true means that this value is assigned to a variable and may not be changed in any way. It can also mean that the value is part of (i.e. is assigned to) a larger data structure.

External Properties

As an alternative to the #ADT directive, which is a comment, the external properties clause in a newtype can be used as container for this information. See the following example:

Example 370: External Properties in a Newtype ––––––––––––––––––

```
NEWTYPE Str STRUCT
  a integer;
  b Boolean;
  c real;
  ADDING LITERALS
    Lit;
  OPERATORS
    Op1 : Str, integer -> Str;
    Op2 : Str, Boolean -> Str;

ALTERNATIVE C;
#ADT (B)
#BODY
 some appropriate C code
ENDALTERNATIVE;

ENDNEWTYPE;
```

––

The #ADT directive, without the /* */ can be placed between ALTERNATIVE C; and ENDALTERNATIVE.

Note:

According to the syntax of SDL, if you have an external properties clause (i.e. alternative - endalternative), you **cannot**, in the same newtype, **have operator diagrams**, axioms, or literal mappings.

More about Operators

For an operator in an abstract data type, not only B (body) or Q (question) may be specified. The following choices are available:

- Q (question)
  This is the default value and specifies that the code generator should generate the interactive routines describe above.
- B (body)
  This specifies that the code generator should generate the heading of the operator or literal function, while the user should supply the body of the function.
- H (heading)
  This specifies that the code generator should neither generate the heading nor the body of the operator or literal function. The user is assumed to supply the necessary code.
- S (standard)
  This is used to indicate that a standard function or operator is available in the target language, which should be used as implementation of the SDL operator (literal). No function heading or function body is generated. In expressions where such an operator is used, no prefix is added to the SDL name during the translation, but the SDL name is used as it is (if no #NAME directive is present).
- P (prefix) or
  I (infix)
  where P is the default value. These letters are used to indicate if the operator should be used as a function or an operator:
  - As an operator: `a+1  a==4  -a`
  - Or as a function call: `sin(a) power(a, 3)`

Note:

As C does not include the possibility to have user defined operators, I (infix) is only adequate together with S (standard).

For each operator one of the letters B, Q, H, S and one of the letters P, I should be supplied, either in a #OP directive, or in a #ADT directive, or as the defaults Q and P; for literals P and I have no meaning.

The purpose of S is straight forward and easy to understand, but H might require some explanation. H means that the code generator will not gen-

erate any code for the operator, which leaves the user with a number of possibilities:

- By not including any code for an operator, the user may skip the code for an unused operator.
- There might already exist external declarations for a number of operators in a `.h` file that should be used instead of the generated headings.

Example 371: Using S (Standard Function or Operator) ––––––––––

Example of usage of S (standard)

```
"+" : integer, real -> real; /*#OP (SI) */
sin : real -> real; /*#OP (SP) */
```

An SDL expression using these operators:

`sin(a + 7.0)` will be translated to: `sin(zh723_a + 7.0)`

––

These examples show how standard functions in the target language can be directly utilized in abstract data types. In C, it is often easiest to use #OP(HP) for such special cases, and implement the operator in the #HEADING section as a C macro transforming the call to the appropriate syntax.

Generic Functions

Type Info Nodes

A generic function can perform a certain task for several different types. To be able to write generic functions, type-specific information for the types must be made available. This type of information could be, for instance, size of the type, component types for structured types and component offsets. This information is provided by the *type info nodes*.

A type info node is a struct that contains information that defines the type. Each type has a corresponding type info node. Each type info node contains two sections. The first section contains a sequence of general components that is identical for all type info nodes. The second section is an individual type-specific sequence of components that defines each unique type.

Every newtype or syntype introduced in SDL will be described by a type info node in the generated C code. For the predefined data types the following type info nodes can be found in `sctpred.h` and `sctpred.c`:

```
extern tSDLTypeInfo ySDL_SDL_Integer;
extern tSDLTypeInfo ySDL_SDL_Real;
extern tSDLTypeInfo ySDL_SDL_Natural;
extern tSDLTypeInfo ySDL_SDL_Boolean;
extern tSDLTypeInfo ySDL_SDL_Character;
extern tSDLTypeInfo ySDL_SDL_Time;
extern tSDLTypeInfo ySDL_SDL_Duration;
extern tSDLTypeInfo ySDL_SDL_PId;
extern tSDLTypeInfo ySDL_SDL_Charstring;
extern tSDLTypeInfo ySDL_SDL_Bit;
extern tSDLTypeInfo ySDL_SDL_Bit_String;
extern tSDLTypeInfo ySDL_SDL_Octet;
extern tSDLTypeInfo ySDL_SDL_Octet_String;
extern tSDLTypeInfo ySDL_SDL_IA5String;
extern tSDLTypeInfo ySDL_SDL_NumericString;
extern tSDLTypeInfo ySDL_SDL_PrintableString;
extern tSDLTypeInfo ySDL_SDL_VisibleString;
extern tSDLTypeInfo ySDL_SDL_Null;
extern tSDLGenListInfo ySDL_SDL_Object_Identifier;
```

For a user-defined type the type info node will have the name

```
ySDL_#(TypeName)
```

Generic Assignment Functions

Each type in SDL has access to an assignment macro `yAssF_typename`. Examples for type Boolean and for a user-defined type A:

```
#define yAssF_SDL_Boolean(V,E,A)  (V = E)

#define yAssF_A(V,E,A)  yAss_A(&(V),E,A)
#define yAss_A(Addr,Expr,AssName) \
   (void)GenericAssignSort(Addr,Expr,AssName,
                           (tSDLTypeInfo *)&ySDL_A)
```

This macro is used in the generated code (and in the kernel) at each location where an assignment should take place. The three macro parameters are:

- **V**: the variable on the left hand side
- **E**: the expression on the right hand side
- **A**: an integer giving the properties of the assignment

This macro will either become an assignment statement in C or a call of an assignment function. An assignment statement will be used if assignment is allowed according to C for the current type and if it has the correct semantics comparing with assignment in SDL.

If assignment is not possible to use, the assign macro will become a call to an assignment function. The basic generic assignment function can be found in `sctpred.c` and `sctpred.h`:

```
extern void * GenericAssignSort(void *, void *,
   int, tSDLTypeInfo *);
```

where:

- The first parameter is the address of the variable on the left hand side.
- The second parameter is the address of the expression on the right hand side.
- The third parameter is the properties of the assignment
- The fourth parameter is the type info node for the actual type.

GenericAssignSort returns the address passed as the first parameter.

The GenericAssignSort function performs three tasks:

1. The old value on the left hand side variable is released, if that is specified in properties of the assignment and if the value contains any pointers.
2. The value is copied from the expression to the variable. If possible this is performed by the function memcpy, otherwise special code depending on the kind of type is executed.
3. The IsAssigned flags are set up for the variable according to the properties of the assignment.

Special treatment of Charstring and instantiations of the Own generator has made it necessary to introduce specific wrapper functions that in their turn call GenericAssignSort for these types:

```
extern void xAss_SDL_Charstring (SDL_Charstring *,
   SDL_Charstring, int);
extern void * GenOwn_Assign (void *, void *, int,
   tSDLTypeInfo *);
```

An GenericAssignSort function must consider the following questions in order to handle the objects correctly.

How should one copy the object?

This is very important because performing the wrong action will lead to memory leaks or access errors. Three different possibilities exist:

- **AC**: always copy the referenced object.
- **AR**: always copy the pointer, i.e reusing the referenced object.
- **MR**: copy pointer if the object is temporary **or** copy object if not temporary.

What should be the status of the new object?

This is a preparation for the next operation on this object so the correct decision can be made according to the first question. Two different possibilities exists:

- **ASS**: an object should become assigned if it is assigned to a variable and needs to be copied in future assignments, i.e corresponds to the values 'V' and 'L' for the first character in a C- string representing the Charstring sort. A typical case is a normal assignment statement in SDL.
- **TMP**: an object should become temporary if it is not assigned to any persistent variable and therefore should not be copied in subsequent assignments, i.e corresponds to the value 'T' for the first character in a C-string representing the Charstring sort. A typical case is a result value from an operator.

What should be done with the old value referenced by the left hand side variable?

Normally free should be performed on the value, as otherwise there would be a memory leak. However, when initializing a variable, no free ought to be performed, as free might be called on a random address. Two different possibilities exists:

  - **FR**: free old value.
  - **NF**: do not free old value.

The third assignment property parameter in the GenericAssignSort function should be given a value according to the ideas given above, preferably using the macros indicated.

```
#define XASS_AC_ASS_FR  (int)25
#define XASS_MR_ASS_FR  (int)26
#define XASS_AR_ASS_FR  (int)28

#define XASS_AC_TMP_FR  (int)17
#define XASS_MR_TMP_FR  (int)18
#define XASS_AR_TMP_FR  (int)20

#define XASS_AC_ASS_NF  (int)9
#define XASS_MR_ASS_NF  (int)10
#define XASS_AR_ASS_NF  (int)12

#define XASS_AC_TMP_NF  (int)1
#define XASS_MR_TMP_NF  (int)2
#define XASS_AR_TMP_NF  (int)4
```

The macro names above are all of the form XASS_1_2_3, where the abbreviations placed at 1, 2, and 3 should be read:

- 1 = AC: always copy
- 1 = MR: may reuse (take pointer if temporary object)
- 1 = AR: always reuse (take pointer)
- 2 = ASS: new object assigned to "variable"
- 2 = TMP: new object temporary
- 3 = FR: call free for old value referred to by variable
- 3 = NF: do not call free for old value

The distinction between all these assignment possibility is only of interest when handling types using or containing pointers.

Generic Equal Functions

Each type in SDL has access to an equal macro `yEqF_typename` and an not equal macro `yNEqF_typename`. Examples for type Boolean and for a user-defined type A:

```
#define yEqF_SDL_Boolean(E1,E2)   ((E1) == (E2))
#define yNEqF_SDL_Boolean(E1,E2)  ((E1) != (E2))

#define yEqF_z3_A(Expr1,Expr2)  yEq_z3_A(Expr1,Expr2)
#define yNEqF_z3_A(Expr1,Expr2)  ( ! yEq_z3_A(Expr1,Expr2) )
#define yEq_z3_A(Expr1,Expr2) \
  GenericEqualSort((void *)Expr1,(void *)Expr2, \
                   (tSDLTypeInfo *)&ySDL_z3_A)
```

These macros are used in the generated code (and in the kernel) at each location where equality tests are needed. The parameters to the equal and not equal macro are the two expressions that should be tested.

If C equal or not equal are not possible to use, the equal macros will become calls to an equal function. The basic generic equal function can be found in `sctpred.h` and `sctpred.h`:

```
extern SDL_Boolean GenericEqualSort(void *, void *,
   tSDLTypeInfo *);
```

where:

- the first two parameters are the addresses to the two expressions to be tested
- the third parameter is the type info node for the actual type.

Special treatment of Charstring and instantiations of the Own generator has made it necessary to introduce specific wrapper functions that in turn calls GenericEqualSort for these types:

```
extern SDL_Boolean xEq_SDL_Charstring
   (SDL_Charstring, SDL_Charstring);
extern SDL_Boolean GenOwn_Equal (void *, void *,
    tSDLTypeInfo *);
```

Generic Free Functions

Each type in SDL that is implemented as a pointer, or that contains a pointer that references to memory that is automatically handled (in principle all pointers except Ref pointers), has access to a corresponding `yFree_typename` function or macro. In the generic function model, this is always a macro.

```
#define yFree_SDL_Charstring(P)  xFree_SDL_Charstring(P)
#define xFree_SDL_Charstring(P) \
   GenericFreeSort(P,(tSDLTypeInfo *)&ySDL_SDL_Charstring)

#define yFree_A(P) \
   GenericFreeSort(P,(tSDLTypeInfo *)&ySDL_A)
```

The yFree macro will always be translated to a call to the function GenericFreeSort.

```
extern void GenericFreeSort (void **, tSDLTypeInfo *);
```

This function takes the address of a variable and a type info node and releases the dynamic memory used by this value contained in the variable.

Generic Make Functions

There are four generic functions constructing values of structured types:

```
extern void * GenericMakeStruct (void *, tSDLTypeInfo *, ...);
extern void * GenericMakeChoice (void *, tSDLTypeInfo *,
   int, void *);
extern void * GenericMakeOwnRef (tSDLTypeInfo *, void *);
extern void * GenericMakeArray  (void *, tSDLTypeInfo *,
   void *);
```

GenericMakeStruct: According to SDL, the Make operator is only available for the struct type. However, in the SDL suite the Make operator, and thus the GenericMakeStruct function, is also available for the Object_identifier type and the instantiations of the generators string, powerset, and bag.

- The void * parameter is the address of a variable where the result should be placed. This value is also returned.
- The tSDLTypeInfo * parameter is the address to the type info node for the type to be created.
- "..." denotes a list of addresses to the values for the components in the struct. All parameters must be passed as addresses (void *) regardless if the component type should be passed as an address or as a value. The only exceptions are the types represented as pointers themselves (Charstring, Ref, Own, ORef, and syntypes of these types), where the pointers are passed, not the addresses of the pointers. In case of an optional field or a field with an initializer, a '0' or '1' is passed to indicate if a value for the component is present or not. If '1' is passed the value follows as next parameter. If '0' is passed no value is present in the actual parameter list.

GenericMakeChoice: This function is used for choice types.

- The first void * parameter is the address of a variable where the result should be placed. This value is also returned.
- The tSDLTypeInfo * parameter is the address to the type info node for the type to be created.
- The int parameter decides which choice component that is present.

- The last void * parameter is the address of the value.

GenericMakeOwnRef: This function is used for instantiations of generators Own and Ref.

- The tSDLTypeInfo * parameter is the address to the type info node for the type to be created.
- The void * parameter is the address to the value that should be assigned to the memory allocated by this function.

GenericMakeArray: This function is used for instantiations of the generators Array, Carray, and GArray.

- The first void * parameter is the address of a variable where the result should be placed. This value is also returned.
- The tSDLTypeInfo * parameter is the address to the type info node for the type to be created.
- The last void * parameter is the address to the value that should be assigned to all components of the array.

Generic Function for Operators in Pre-defined Generators

The generic function for the operators in the pre-defined generators follow the general rules for operators with a few exceptions:

- a type info node is needed as a parameter, as the C function can handle all instantiations of a certain generator.
- parameters of generator parameter types (component and index types for example) must in many cases be passed as addresses, as the properties of these types are not known.

General array

```
extern void * GenGArray_Extract (xGArray_Type *, void *,
   tSDLGArrayInfo *);
extern void * GenGArray_Modify  (xGArray_Type *, void *,
   tSDLGArrayInfo *);
```

- Parameter 1: The array
- Parameter 2: The index value passed as an address

- Parameter 3: The type info node
- Result: The address of the component

Powerset

Generic functions available for powersets with a simple component type. The powerset is represented a sequences of bits (unsigned char[Appropriate_Length]).

```
#define GenPow_Empty(SDLInfo,Result) \
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern SDL_Boolean GenPow_In (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern void * GenPow_Incl (int, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern void * GenPow_Del (int, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern void GenPow_Incl2 (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern void GenPow_Del2 (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern SDL_Boolean GenPow_LT (xPowerset_Type *,
   xPowerset_Type *, tSDLPowersetInfo *);
extern SDL_Boolean GenPow_LE (xPowerset_Type *,
   xPowerset_Type *, tSDLPowersetInfo *);
extern void * GenPow_And (xPowerset_Type *, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern void * GenPow_Or (xPowerset_Type *, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern SDL_Integer GenPow_Length (xPowerset_Type *,
   tSDLPowersetInfo *);
extern int GenPow_Take (xPowerset_Type *, tSDLPowersetInfo *);
extern int GenPow_Take2 (xPowerset_Type *, SDL_Integer,
   tSDLPowersetInfo *);
```

- **Parameter of type int in GenPow_In, GenPow_Incl, GenPow_Del, GenPow_Incl2, GenPow_Del2**: A component value.
- **Result of type int in GenPow_Take, GenPow_Take2**: A component value.
- **Parameters of type tSDLPowersetInfo ***: The type info node.
- **Parameters of type xPowerset_Type * after the type info node**: The address where the result should be stored. This address is returned by the function.
- **Other xPowerset_Type * parameters**: Powerset in parameters.

Bag and General Powerset

The following generic functions are available for bags and powersets with complex component type. These types are represented as linked lists in C.

```
#define GenBag_Empty(SDLInfo,Result) \
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenBag_Makebag (void *, tSDLGenListInfo *,
   xBag_Type *);
extern SDL_Boolean GenBag_In (void *, xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_Incl (void *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void * GenBag_Del (void *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void GenBag_Incl2 (void *, xBag_Type *,
   tSDLGenListInfo *);
extern void GenBag_Del2 (void *, xBag_Type *,
   tSDLGenListInfo *);
extern SDL_Boolean GenBag_LT (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *);
extern SDL_Boolean GenBag_LE (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_And (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void * GenBag_Or(xBag_Type *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern SDL_Integer GenBag_Length (xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_Take (xBag_Type *, tSDLGenListInfo *,
   void *);
extern void * GenBag_Take2 (xBag_Type *, SDL_Integer,
   tSDLGenListInfo *, void *);
```

- **Parameter of type int in GenBag_Makebag, GenBag_In, GenBag_Incl, GenBag_Del, GenBag_Incl2, GenBag_Del2**: The address of the component value.
- **Result of type int in GenBag_Take, GenBag_Take2**: The address of the component value.
- **Parameters of type tSDLGenListInfo ***: The type info node.
- **Parameters of type xBag_Type * after the type info node**: The address where the result should be stored. This address is returned by the function.
- **Parameters of type void * after the type info node**: The address where the result should be stored. This address is returned by the function.
- **Other xBag_Type * parameters**: Bag/Powerset in parameters.

String

The following Generic functions are available for String instantiations. A String is implemented as a linked list.

```
#define GenString_Emptystring(SDLInfo,Result) \
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenString_MkString (void *, tSDLGenListInfo *,
   xString_Type *);
extern SDL_Integer GenString_Length (xString_Type *,
   tSDLGenListInfo *);
extern void * GenString_First (xString_Type *,
   tSDLGenListInfo *, void *);
extern void * GenString_Last (xString_Type *,
   tSDLGenListInfo *, void *);
extern void * GenString_Concat (xString_Type *,
   xString_Type *, tSDLGenListInfo *, xString_Type *);
extern void * GenString_SubString (xString_Type *,
   SDL_Integer, SDL_Integer, tSDLGenListInfo *,
   xString_Type *);
extern void GenString_Append (xString_Type *, void *,
   tSDLGenListInfo *);
extern void * GenString_Extract (xString_Type *, SDL_Integer,
   tSDLGenListInfo *);
```

- **Parameter of type void * in GenString_MkString, GenString_Append**: Address of component value.
- **Parameter of type void * or xString_Type * after type info node**: The address where the result should be stored. This address is returned by the function.
- **Parameters of type tSDLGenListInfo ***: The type info node.
- **Other parameters**: According to SDL definition of parameters.

Limited String

Generic functions available for limited strings, i.e. strings with #STRING directive giving a max size of the string. These strings are implemented as an array in C.

```
#define GenLString_Emptystring(SDLInfo,Result) \
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenLString_MkString (void *, tSDLLStringInfo *,
   xLString_Type *);
#define GenLString_Length(ST,SDLInfo) (ST)->Length
extern void * GenLString_First (xLString_Type *,
   tSDLLStringInfo *, void *);
extern void * GenLString_Last (xLString_Type *,
   tSDLLStringInfo *, void *);
extern void * GenLString_Concat (xLString_Type *,
   xLString_Type *, tSDLLStringInfo *, xLString_Type *);
extern void * GenLString_SubString (xLString_Type *,
   SDL_Integer, SDL_Integer, tSDLLStringInfo *,
   xLString_Type *);
extern void GenLString_Append (xLString_Type *, void *,
```

```
   tSDLLStringInfo *);
extern void * GenLString_Extract (xLString_Type *,
   SDL_Integer, tSDLLStringInfo *);
```

- **Parameter of type void * in GenLString_MkString, GenString_Append**: Address of component value.
- **Parameter of type void * or xLString_Type * after type info node**: The address where the result should be stored. This address is returned by the function.
- **Parameters of type tSDLLStringInfo ***: The type info node.
- **Other parameters**: According to SDL definition of parameters.

More about Abstract Data Types

Including Type Definitions

> **Note:**
>
> Use the features presented in this section with care. The features were developed early in the SDL suite history. Now in principle every data type in C can be expressed in SDL as well. Therefore, the recommended method is to write the types in SDL or to translate C types to SDL using the cpp2sdl tool.
>
> History has also shown that it has been difficult to keep full backward compatibility for these features and at the same time improve the performance of the generated code. This of course comes from that these features is highly dependent on the way code is generated.

In this subsection, the inclusion of a type definition in the target language for an abstract data type will be described. When this facility is used, it is necessary to specify how to perform assignment, test for equal, assign default values, and so on, as it is not possible to generate when the type definition is not known (not generated). All this information is given in the #ADT directive, which has the following structure:

```
/*#ADT
  (T(x) A(x) E(x) F(x) K(x) X(x) M(x) W(x) R(x)
   xy 'file name')
#TYPE
C code
#HEADING
C code
#BODY
```

```
C code
*/
```

where each `x` on the first line should be replaced by one of the characters B, H, Q, S, or G. Replace `y` by P or I. The interpretation of these characters is similar to the their interpretation for operators.

B	Body
H	Heading
Q	Question
S	Standard
G	Generate
P	Prefix
I	Infix

The reason why G (generate) is not allowed for operators or literals is of course that it would mean to generate the implementation of the operators from the axioms, which is, at least in the general case, an impossible task. For an operator defined in an operator diagram, G is assumed independently of what the user specifies.

The specifications, given in ADT directives, of how to generate code for type definition, assignment, test for equal, default values, and free function should be interpreted according to the table below.

Type Definition

First the actual type definition. The entry - should be interpreted as if no specification is given for T.

Type	**Interpretation**
T(G)	Generate type definition from SDL sort
T(B)	Do not generate type definition. Assume the type should be “passed as value” to operators
T(BV)	Do not generate type definition. Assume the type should be “passed as address” to operators

Type	Interpretation
T	Same as T(B)
-	Same as T(G)

Assignment

It is possible to select how assignments should be performed for values of the type. Note that all generated assignments will be of the form:

```
yAssF_#(SortName)(....);
```

The yAssF_#(SortName) is a macro either implemented as assignment or as a call to the yAss_#(SortName) function (if such function is to be used), i.e as:

```
#define yAssF_#(S)(V,E,A) V = E
#define yAssF_#(S)(V,E,A) yAss_#(S)(&(V),E,A)
```

Type	Interpretation
A(B)	Use and generate heading, but not body, of yAss_#(S)
A(H)	Use, but generate no code for yAss_#(S)
A(G)	• If the type definition **is** generated: – Use = if possible. – Otherwise use the GenericAssignSort function • If type definition **is not** generated (T, T(B)): – Use =
A(S)	Use =
A	Same as A(B)
-	Same as A(G)

If you define your own assign function, it must be implemented as a function, as the address of the function will be stored (in the type info node for the data type) so GenericAssignSort can call it to handle sub-components of this type. An assign function has the following heading:

```
void yAss_#(SortName)
  (#(SortName) *yVar,
   #(SortName) *yExpr,
   int          AssType)
```

It should assign the value passed as second parameter to the variable passed as first parameter. If the type that is to be assigned contains any pointers the assign function is a bit complicated to write in order to avoid access errors and memory leaks. See the discussion about the AsType parameter to GenericAssignSort in "Generic Assignment Functions" on page 2624.

Equal Test

It is possible to select how test for equality should be performed for values of the type. Note that all generated equal tests will be of the form:

```
yEqF_#(SortName)(....);
```

The yEqF_#(SortName) is a macro either implemented as C equal or as a call to the yEq_#(SortName) function (if such function is to be used), i.e as:

```
#define yEqF_#(S)(E1,E2) E1 == E2
#define yEqF_#(S)(E1,E2) yEq_#(S)(E1,E2)
```

The /= operator is represented by the macro

```
#define yNEqF_#(S)(E1,E2) (! yEqF_#(S)(E1,E2) ).
```

Type	Method
E(B)	Use and generate heading, but not body, of yEq_#(S)
E(H)	Use but generate no code for `yEq_#(S)`
E(G)	• If the type definition **is** generated: – Use `==` if possible – Otherwise use the GenericEqualSort function. • If the type definition **is not** generated – Use `==`
E(S)	Use `==`
E(Q)	Use and generate an equal function that asks for the result of the test (same as Q for operators).
E	Same as E(B)
-	Same as E(G)

If you define your own equal function, it must be implemented as a function, as the address of the function will be stored (in the type info node for the data type) so GenericEqualSort can call it to handle sub-components of this type. An equal function has the following heading:

```
SDL_Boolean yEq_#(SortName)
  (#(SortName) *yExpr1,
   #(SortName) *yExpr2);
```

It should return `true` or `false` depending on if the two values passed as parameters are equal or not. If the parameters contain pointers it might be necessary to free these values, please see the discussion on general parameters to operators in"Other Types Containing Pointers" on page 2620.

Free of Dynamic Memory

This section describes how dynamic memory (if used for the type) will be released for reuse when it is no longer needed.

Type	Interpretation
F(B)	Generate heading, but no body of the free function `yFree_#(SortName)`.
F(H)	Generate neither heading nor body of the free function.
F(S)	Use the function `GenericFreeSort`
F	Same as F(B)
-	Do not use free function.

If you define your own free function, it must be implemented as a function, as the address of the function will be stored (in the type info node for the data type) so GenericFreeSort can call it to handle subcomponents of this type. A free function should have the following prototype

```
void yFree_#(SortName) (void **yVar)
```

The function should take the address to a pointer, return the allocated memory to the pool of available memory and assign 0 to the pointer.

Extract! and Modify!

This entry specifies how component selection (struct components, array components for example) should be performed. In SDL a component can be selected in two ways:

```
Variable ! Component
Variable (Index)
```

An Extract operation can be generated in four ways:

```
Variable.Component       used for struct and #UNIONC
Variable.U.Component     used for #UNION and choice
Variable.A(Index)        used for array
yExtr_SortName(Variable,Expr)
```

The last version, the Extract function, is used for all other cases.

Type	Interpretation
X(B)	Use Extract function
X(G)	Use component selection according to table above.
X	Same as X(B)
-	Same as X(G)

A Modify operation can in the same way be generated in four ways:

```
Variable.Component       used for struct and #UNIONC
Variable.U.Component     used for #UNION and choice
Variable.A(Index)        used for array
(* yAddr_SortName((&Variable), Expr))
```

The last version, the Addr function, is used for all other cases.

Type	Interpretation
M(B)	Use Addr function
M(G)	Use component selection according to table above.
M	Same as M(B)
-	Same as M(G)

Read and Write Function

The write function is used by the monitor system to write values of the type.

Type	Interpretation
W(B)	Generate heading but not the body of a write function.
W(H)	Generate neither heading nor body of a write function, but assume that the user has provided such a function.
W(S)	Values of this type are to be printed as a HEX string. No write function is assumed to be present.
W	Same as W(B)
-	Same as W(S)

The read function is used by the monitor system to read values of the type.

Type	Interpretation
R(B)	Generate heading but not the body of a read function.
R(H)	Generate neither heading nor body of a read function, but assume that the user has provided such a function.
R(S)	Values of this type are to be read as a HEX string. No read function is assumed to be present.
R	Same as R(B)
-	Same as R(S)

In order to examine variable values for variables that are of a sort that is implemented in C, i.e. has #ADT(T) in its ADT directive, it is necessary to implement a write function. Otherwise the value can only be presented as a HEX string. Note that the run-time kernel can automatically handle all SDL sorts for which the code generator generates the C type definition. A write function should look like:

```
extern char * yWri_SortName (void * Value)
```

Given the address of a value of the type SortName, this function should return a char *, i.e. a character string, containing the value represented

in a printable form. This character string is the string that will be printed by the monitor, when it needs to print a value of this type. To implement the write function it is not uncommon that a static char array is needed.

Note:

The following two considerations when it comes to write and read functions:

- The read and write functions and any help variables and help functions ought to be surrounded by

  ```
  #ifdef XREADANDWRITEF
  #endif
  ```

  to be removed if read and write functions are not needed.

- The string format used to represent value of a type should be the same in the read and the write function. **Otherwise communicating simulations will not work** if this type is passed as parameter between the systems.

The function xWriteSort, which is part of the run-time kernel can be useful when implementing Write functions.

```
extern char * xWriteSort (
  void        *In_Addr,
  xSortIdNode  SortNode);
```

The xWriteSort function takes the address of a value to be printed, and a pointer to a xSortIdNode and returns the given value as a string. This function is typically useful if the sort we are implementing a write function for contains one or several components of sorts defined in SDL.

Read Function

In order to assign new values to variables that are of a sort that is implemented in C, i.e. has #ADT(T) in its ADT directive, it is necessary to implement a read function. Otherwise the value can only be read as a HEX string. Note that the run-time kernel can automatically handle all SDL sorts for which the code generator generates the C type definition. A read function should look like:

```
extern int yRead_SortName (void * Result)
```

A read function is given an address to store the value that is read. It should return 1 if the read operation was successful. Otherwise, 0 should be returned and `Result` should be unchanged.

There are some suitable functions in the run-time kernel which can help you when you are implementing a read function. Basically the function xScanToken described below is a tokenizer that transforms sequences of characters to tokens. This function returns tokens according to the following enum type:

```
typedef enum {
  xxId,                 /* identifiers, numbers   */
  xxString,             /* SDL Charstring literal */
  xxSlash,              /* /  */
  xxColon,              /* :  */
  xxMinus,              /* -  */
  xxPlus,               /* +  */
  xxStar,               /* *  */
  xxComma,              /* ,  */
  xxSemicolon,          /* ;  */
  xxArrowHead,          /* ^  */
  xxLPar,               /* (  */
  xxRPar,               /* )  */
  xxLParDot,            /* (. */
  xxRParDot,            /* .) */
  xxLParColon,          /* (: */
  xxRParColon,          /* :) */
  xxDot,                /* .  */
  xxLBracket,           /* [  */
  xxRBracket,           /* ]  */
  xxLCurlyBracket,      /* {  */
  xxRCurlyBracket,      /* }  */
  xxAt,                 /* @  */
  xxQuaStart,           /* << */
  xxQuaEnd,             /* >> */
  xxLT,                 /* <  */
  xxLE,                 /* <= */
  xxGT,                 /* >  */
  xxGE,                 /* >= */
  xxEQ,                 /* =  */
  xxNE,                 /* /= */
  xxQuestionMark,       /* ?  */
  xx2QuestionMark,      /* ?? */
  xxExclMark,           /* !  */
  xxSystem,             /* system       */
  xxPackage,            /* package      */
  xxBlock,              /* block        */
  xxProcess,            /* process      */
  xxProcedure,          /* procedure    */
  xxOperator,           /* operator     */
  xxSubstructure,       /* substructure */
  xxChannel,            /* channel      */
  xxSignalroute,        /* signalroute  */
  xxType,               /* type         */
  xxNull,               /* null         */
  xxEnv,                /* env          */
  xxSelf,               /* self         */
  xxParent,             /* parent       */
```

```
  xxOffspring,        /* offspring    */
  xxSender,           /* sender       */
  xxEoln,             /* end of line */
  xxEOF,              /* end of file */
  xxErrorToken        /* used to indicate error */
} xxToken;
```

Function xScanToken

The function xScanToken:

```
extern xxToken xScanToken ( char * strVar);
```

reads the next token from input (stdin or Simulator UI) and returns the type of the next token as function result. If the token is xxId or xxString the strVar parameter will contain the identifier, number, or string. The size of the char[] parameter passed as actual parameter should be large enough to store the possible values. If some other token was found, no information is stored in `strVar`.

xUngetToken

Sometimes it is necessary to look-ahead to determine how to handle the current token. Using the function xUngetToken below it is possible return one token to the input. Note that both parameters must have the values obtained from xScanToken.

```
extern void xUngetToken (
  xxToken Token,
  char  * strVar);
```

The functions below can also be useful while implementing Read function. xPromptQuestionMark is suitable to obtain prompt in a similar way as for SDL defined sorts, while xReadOneParameter can be used to read element for element in a list, separated by commas and terminated either by ".)" or "]". The function xReadSort is similar to xWriteSort and can be used to read a component in the treated sort.

xPromptQuestionMark

```
extern xxToken xPromptQuestionMark (
  char  * Prompt,
  char  * QuestionPrompt,
  char  * strVar);
```

The function result and the parameter strVar behave in the same way as for the function xScanToken (see above). The parameter Prompt is the prompt that should be used. This string has to start with a ' ', i.e. a space. To conform with other built-in read function, the Prompt parameter

should be: " (SortName) : " (note the ending space colon space). The QuestionPrompt parameter should either be identical to the Prompt parameter, or be null, i.e. (char *)0. If QuestionPrompt is null, the xPromptQuestionMark function will return xxEoln if a end-of-line is found. If QuestionPrompt is not null, the xPromptQuestionMark function will print the QuestionPrompt, and continue to read. Normally QuestionPrompt should be equal to Prompt in a simple data type, while it should be null in a structured data type.

Example 372: ADT Example, Byte Type ––––––––––––––––––––––––

This example is taken from the ADT byte (see "Abstract Data Type for Byte" on page 3176 in chapter 63, *The ADT Library*). The byte type should be read and printed using HEX format.

```
#ifdef XREADANDWRITEF
static char yCTmp[20];

extern int yRead_byte( void  *Result )
{
  unsigned temp;
  xxToken Token;

  Token = xPromptQuestionMark(" (byte) : ",
     " (byte) : ", yCTmp);

  if (Token==xxId && sscanf(yCTmp, "%X", &temp)>=1) {
    *(byte *)Result = (byte)temp;
    return 1;
  }
  xPrintString("Illegal byte value\n");
  return 0;
}

extern char *yWri_byte( void  * C)
{
  sprintf(yCTmp, "%0.2X", *(byte *)C);
  return yCTmp;
}
#endif
```

––

Example 373: ADT Example, Struct Write Functions ––––––––––––

This is an example of how the read and write functions for a struct with two components can look. The monitor system can handle reading of writing of struct values automatically, so please see this just as an example.

```
newtype struct1 /*#NAME 'struct1' */ struct
  a,b Integer;

/*#ADT(W)
#BODY
#ifdef XREADANDWRITEF
static char CTemp[500];

char * yWri_struct1 (void *In_Addr)
{
  strcpy(CTemp, "(. ");
  strcat(CTemp, xWriteSort((void *)
    (&((struct1 *)In_Addr)->a), xSrtN_SDL_Integer) );
  strcat(CTemp, ", ");
  strcat(CTemp, xWriteSort((void *)
    (&((struct1 *)In_Addr)->b), xSrtN_SDL_Integer) );
  strcat(CTemp, " .)");
  return CTemp;
}
#endif
*/
endnewtype;
```

More about #ADT

When generate is specified for a function, the code generator might decide not to generate the heading of the function, as in some cases it is not needed.

All code that is not generated is assumed to be included by the user in the #TYPE, #HEADING and #BODY sections in the #ADT directive.

Another name for an assign function, equal function and so on may be used, by including the desired name within quotes together with the generation options in the #ADT directive.

If, for example, the name of a certain assign function should be AssX, this can be obtained by specifying: A(B 'AssX') for the assign function. This name will then be used throughout the generated code, both in generated declaration and at the places where the function is called. The name should be last in the specification for the actual function.

An include statement may be generated together with or replacing the type definition by giving a file name within quotes last in the specification part of the #ADT directive, immediately before the first section with code.

Example 374: Including a File in ADT ––––––––––––––––––––––––––

If the directive

```
/*#ADT (T(B) A(S) E(S) 'file name') */
```

is used, the following include statement will be generated:

```
#include "file name"
```

––

> **Note:**
>
> Turning off the generation of the objects contained in the include file must be performed by the user.

Directive #REF

> **Note:**
>
> This directive is provided only for backward compatibility. The SDL suite now supports in/out parameters for operators, which serves exactly the same purpose. In/out parameter is an SDL-2000 extension and is supported also in operator diagrams.

The directive #REF can be used to specify that the **address**, **not the value**, of a variable should be passed as parameter to an ADT operator, as it is defined in SDL. This feature cannot be used for operators defined in operator diagrams (the directive will be ignored for such operators).

The #REF directive is used as shown in the example below.

Example 375: Including a File in ADT ––––––––––––––––––––––––––

```
operators
  eq1 : Integer, Integer -> Integer;
  eq2 : Integer/*#REF*/, Integer/*#REF*/ -> Integer;
```

The headings for these two operator will become in ANSI-C syntax (ignoring prefixes)

```
extern SDL_Integer eq1 (SDL_Integer P1,
                        SDL_Integer P2);
extern SDL_Integer eq2 (SDL_Integer *P1,
                        SDL_Integer *P2);
```

––

This feature can be used to optimize parameter passing to operators. The directive, however, also imposes the restriction that the actual parameters must be a variable or a formal parameter (see Example 376 below).

This is checked by the code generator. A #REF directive does not in any way effect the way a operator call should be implemented in SDL. It is the responsibility of the code generator to generate the proper actual parameters in C.

Example 376: Including a File in ADT

With the ADT in the previous example the following operator call is valid:

```
eq1(sVar, (. 1, 2, 3, 4 .) )
```

The same call of `eq2` would not be valid as the second parameter is not a variable or a formal parameter.

Generators

The Cadvanced/Cbasic SDL to C Compiler handles all the predefined generators in SDL, i.e. Array, String, Powerset, and Bag. It is also possible for a user to write his own generators and instantiate them in newtypes. However, the behavior of a user defined generator has to be specified completely by the user. This is performed in a somewhat extended #ADT directive placed just before the endgenerator keyword. These extensions are described below.

There are three additional sections in the directive, apart from #TYPE, #HEADING, and #BODY. These are #INSTTYPE, #INSTHEADING, and #INSTBODY. The inline C code in #TYPE, #HEADING, and #BODY is placed at the point of the generator, i.e. it is generated once. The contents of #INSTTYPE, #INSTHEADING, and #INSTBODY is inserted at each instantiation of the generator, i.e. in each newtype defined using the generator.

In the #INSTTYPE, #INSTHEADING, and #INSTBODY it is possible to use # followed by a number to access the information given in the generator instantiation:

- #0 means the name of the newtype in the instantiation
- #1 and ##1 is the first actual generator parameter
- #2 and ##2 is the second actual generator parameter
- and so on.

#1 and ##1 are equal, except when the corresponding actual generator parameter is a struct (or union). In that case, assuming the SDL struct:

```
newtype aaa struct
  a, b integer;
endnewtype;
```

which will be generated as

```
typedef struct aaa_s {
  SDL_Integer a;
  SDL_Integer b;
} aaa;
```

#1 will become `struct aaa_s` (or `union aaa_s` if a union), while ##1 will become `aaa`.

Example 377: Example of User Defined Generator ----------

```
GENERATOR Ref (TYPE Itemsort)
  LITERALS
    Null;                              /*#OP(S)*/
  OPERATORS
    Ref2VStar : Ref -> VoidStar;     /*#OP(HP)*/
  DEFAULT Null;
/*#ADT()
#INSTTYPE
typedef #1 *#0;
#INSTHEADING
#define #(Null)()       0
#define #(Ref2VStar)(P) ((#(VoidStar))P)
*/
ENDGENERATOR Ref;
```

Note the usage of #INSTTYPE and #INSTHEADING in the example above. The code in these section will be inserted in each newtype defined with this generator. For example, in a newtype:

```
NEWTYPE p Ref(Integer)
ENDNEWTYPE;
```

The #INSTTYPE section will become:

```
typedef SDL_Integer *p;
```

Directives to the Cadvanced/Cbasic SDL to C Compiler

Syntax of Directives

The Cadvanced/Cbasic SDL to C Compiler recognizes a number of directives given mainly in SDL comments. The #ADT, #OP, #UNION, and #REF directives used in abstract data types are examples of such directives. The directives #ADT and #OP were described in the section "Implementation of User Defined Operators" on page 2609, "Union" on page 2598, and "Directive #REF" on page 2646, in connection with abstract data types and are not further discussed here.

A directive has the general structure:

1. The start of comment character: /*
2. A '#' character.
3. The directive name.
4. Possible directive parameters given in free syntax. That is, spaces and carriage returns are allowed here.
5. The end of comment characters */.

Upper and lower case letters are considered to be equal in directive names.

Example 378: #OP Directive ––––––––––––––––––––––––––––––––

Take as an example the directive:

```
/*#OP (B) */.
```

This comment will be recognized as a directive only if no other character is inserted in the sequence `/*#OP`. After this part, spaces and carriage returns may be inserted freely.

––

Selecting File Structure for Generated Code – Directive #SEPARATE

The purpose of the separate generation feature is to specify the file structure of the generated program. Both the division of the system into a number of files and the actual file names can be specified. There are two ways this information can be given.

- Normally this information is set up in the Organizer, using the *Edit Separation* command, see "Edit Separation" on page 136 in chapter 2, *The Organizer*. Here file names for the generated files can also be specified. In the *Make* dialog in the Organizer (see "Make" on page 119 in chapter 2, *The Organizer*) it is possible to select full separate generation, user-defined separate generation, or no separate generation.

- For an SDL/PR file that is used as input when running the SDL Analyzer as a stand-alone tool, the same information can be entered by #SEPARATE directives directly introduced in the SDL program. Full separate generation, user-defined separate generation, or no separate generation can be set up in the command interface of a stand-alone Analyzer, see *"Set-Modularity" on page 2421 in chapter 55, The SDL Analyzer*.

The Cadvanced/Cbasic SDL to C Compiler can generate a separate file for:

- system (always separate)
- package (always separate)
- system type
- block
- block type
- process
- process type
- service
- service type
- procedure

Note:

Instantiations cannot be separated, i.e. an instance of a block type cannot be generated on a file of its own.

If #SEPARATE directives are used, they should be placed directly after the first semicolon in the system, block, process, or procedure heading; see the following example.

Example 379: #SEPARATE Directive ––––––––––––––––––––––––

```
system S; /*#SEPARATE 'filename' */
block B; /*#SEPARATE */
process type P1 inherits PType; /*#SEPARATE */
process P2 (1, ); /*#SEPARATE */
procedure Q; /*#SEPARATE */
```

––

In the example above the two versions of separate directive, with or without file name, are shown. As can be seen a file name should be enclosed between quotes. The code generator will append appropriate extensions to this name when it generates code.

If no file name is given in the directive, the name of the system, block, process, or procedure will be used to obtain a file name. In such case the file name becomes the name of the unit with the appropriate extension (`.c .h`) depending on contents. The file name is stripped from characters that are not letters, digits or underscores.

The possibility to set up full, user-defined, or no separation in the Organizer's *Make* dialog and in the user interface of a stand-alone Analyzer (see *"The Analyzer Command-Line UI" on page 2404 in chapter 55, The SDL Analyzer*), can be used, in simple manner, to select certain default separation schemes. This setting will be interpreted in the following way:

- *No* separation.
  The system and each package will be generated on a separate file.
- *User defined* separation.
  The system, each package, and each unit that the user has specified as separate will become a separate file.
- *Full* separation.
  The system, each package, each block, block type, process, process type, service, and service type will become a separate file. Note that even in this case a procedure is separate only if the user has specified it as separate.

Independently if *No*, *User defined*, or *Full* separation has be selected, the code generator will use the file name specified in the *Edit Separa-*

tion dialog or the #SEPARATE directive, for a file that is to be generated.

An Example of the Usage of the Separate Feature

In the following example a system structure and the #SEPARATE directives are given. The same information can easily be set up in the Organizer as well. This example is then used to show the generated file structure depending on selected generation option.

Example 380: #SEPARATE Directive ––––––––––––––––––––––––––

```
system S; /*#SEPARATE 'Sfile' */
  block B1; /*#SEPARATE */
    process P11; /*#SEPARATE 'P11file' */
    process P12;
  block B2;
    process P21;
    process P22; /*#SEPARATE */
```

Note that #SEPARATE directives can only be used in SDL/PR files. Normally this information is given in the Organizer.

––

Applying Full Separate Generation

If *Full* separate generation is selected then the following files will be generated:

Sfile.c	Sfile.h
B1.c	B1.h
P11file.c	
P12.c	
B2.c	B2.h
P21.c	
P22.c	

The `.c` files contain the C code for the corresponding SDL unit and the `.h` files contain the module interfaces.

Applying Separate Generation

If *User defined* separate generation is selected then the following files will be generated:

`Sfile.c`	`Sfile.h`	Contains code for units S, B2, P21
`B1.c`	`B1.h`	Contains code for units B1, P12
`P11file.c`		Contains code for unit P11
`P22.c`		Contains code for unit P22

The user defined separate generation option thus makes it possible for a user to completely decide the file structure for the generated code. The comments on files and extensions given above are, of course, also valid in this case.

Applying No Separate Generation

If the separation option *No* is selected, only the following file will be generated:

`Sfile.c`		Contains code for all units

The comments on files and extensions earlier are valid even here.

Guidelines

Generally a system should be divided into manageable pieces of code. That is, for a **large system full separate generation** should be used, while for a **small system no separate generation** ought to be used. The possibility to regenerate and recompile only parts of a system usually compensate for the overhead in generating and compiling several files for a large system.

Note:

A file name has to be specified, in the Organizer *Edit Separation* command, see "Edit Separation" on page 136 in chapter 2, *The Organizer*, or in the #SEPARATE directive, if two units in the system have the same name in SDL and should both be generated on separate files, otherwise the same file name will be used for both units.

Accessing SDL Names in C Code – Directive #SDL

When writing C code that is to be included in a generated program it is often necessary to refer to names of objects defined in SDL. The name of an SDL object is, however, transformed when it is translated to C. A prefix, which is a sequence of characters, is added to the SDL name to make the C name unique in the C program. Furthermore, all characters in SDL name which are not allowed in a C name are removed. The prefixes are calculated by looking at the structure of definitions in the actual scope and in all scopes above. This means that adding a declaration at the system level might change all prefixes in blocks and processes contained in the system. As a consequence it is almost impossible to know the prefix of an object in advance.

To be able to write C code and use the name of SDL objects in that code, the Cadvanced/Cbasic SDL to C Compiler provides the directive #SDL which is used in C code to translate an SDL name to the corresponding C name.

The syntax of the #SDL directive is as follows:

```
#SDL (SDL name)
```

or

```
#SDL (SDL name, entity class name)
```

There is also a short form for the directive. No characters are allowed between the # character and the left parentheses in this form:

```
#(SDL name)
```

or

```
#(SDL name, entity class name)
```

Replace `SDL name` with the name of an object in the SDL definition and `entity class name` by any of the following identifiers (upper and lower case letters are considered to be equal):

block blockinst blocksubst blocktype channel channelsubst connect formalpar gate generator label literal newtype	operator package predef procedure process processinst processtype remoteprd remotevar service serviceinst servicetype	signal signallist signalroute sort (= newtype) state synonym syntype system systemtype timer variable view

This list contains all entity classes, which means that not all of the entries are relevant for practical use. When a #SDL directive is found in included C code, the code generator first identifies what SDL object is referred to and then replaces the directive by the C name for that object. The search for the SDL object starts in the current scope (the scope where the C code is included), and follows the scope hierarchy outward to the system definition, until an appropriate SDL object is found. An appropriate SDL name is considered to be found if it has the specified name and is in the specified entity class. If no entity class name is given the search is performed for all entity classes.

Note:

In types, especially in block types, #SDL should be used with care. The reason is that some of the objects in a block type are generated for each instantiation of the block. A #SDL directive on such an object might lead to overloading of names in C. Sensitive objects are processes, process instantiations, signal routes, channels, remote definitions.

The table in the subsection "SDL Predefined Types" on page 2588 gives the direct translation between an SDL name and the corresponding C name or expression. For these names the #SDL directive should not be used.

Including C Code in Task – Directive #CODE

The user's own C code may be included in tasks by using the #CODE directive. This directive has the following syntax:

```
/*#CODE
C code that should
be included in
generated code */
```

Type the directive name on the first line and the C code **on the following lines** up to the end of comment symbol. Note that text on the same line as the #CODE directive are not handled.

A #CODE directive can be placed:

- Before the first assignment statement or informal text
- Immediately preceding or just following the comma that separates two assignment statements or informal texts
- After the last assignment statement or informal text
- Immediately after the ending semicolon (;) of a task (this position is only available in SDL/PR).

The C code in the directives is textually included in the generated code at the position of the directive. If, for example, a code directive is placed between two assignment statements, the code in the directive is inserted between the translated version of the assignment statements.

> **Note:**
>
> The Cadvanced/Cbasic SDL to C Compiler handles the C code in directives as text and performs **no check that the code is valid C code**.

The code directive is included as a facility in the code generator to provide experienced users an escape possibility to the target language C. This increases the application range of the code generator.

An example of a possible use of the code directive is: An algorithm for some computation, which in the SDL description is only indicated as a task with an informal text, could be implemented in C. In this case the directive #SDL described in the previous subsection will probably become useful to access variables and formal parameters defined in SDL.

Some general hints on how to write C code that can be included into a simulation program, especially when charstrings or sorts containing charstrings as components are used, can be found in the last part of the section "Implementation of User Defined Operators" on page 2609.

Unfortunately it is not possible to have C comments within the code that is included in any directive, as SDL and C use the same symbols for start and end of comments. See also Example 360 on page 2613 which illustrates the possibility to use the C macro COMMENT.

#CODE directives in compound statements

#CODE directives are recognized after a semicolon that ends a statement of one of the following kinds:

- Output statement
- Create statement
- Set statement
- Reset statement
- Export statement
- Return statement
- Call statement
- Assignment statement
- Break statement
- Continue statement
- Empty statement

Example 381 ----------

```
{
  ;  /*#CODE
       #(i) = #(i)+1; */
  i := i+2;  /*#CODE
       #(i) = #(i)+3; */
}
```

This example contains first an empty statement followed by a directive and then an assignment followed by a directive. The empty statement can, as above, be used to insert #CODE directives at places that otherwise would not be possible, like at the beginning of a compound statement or directly after a compound statement.

Note that the code in the #CODE directive is associated to the statement just before the directive and is included in the scope of that statement.

Example 382 ----------

```
{
  if (true)
    i := i+4; /*#CODE
       #(i) = #(i)+5; */
}
```

In this case the code in the directive is included in the "if-part" of the if statement, just like the assignment it is associated with. This will be treated as:

```
  if (true) {
    i := i+4;
    i := i+5;
  }
```

To put the code directly after the if statement the following structure, with an empty statement after the if statement, can be used:

```
{
  if (true)
    i := i+4;
  ; /*#CODE
       #(i) = #(i)+5; */
}
```

Including C Declarations – Directive #CODE

The #CODE directive can also be used to include C declarations; for example types, variables, functions, #define, and #include in the declaration parts of the C program. This version of the code directive has the following structure:

```
/*#CODE
#TYPE
C code containing:
Types and variables
#HEADING
C code containing:
Extern or static declarations of functions
#BODY
C code containing:
Bodies of functions
*/
```

The separation of functions into HEADING and BODY sections serves the same purpose as in the #ADT directive, see "Implementation of User Defined Operators" on page 2609.

Code directives to include C declarations may, generally speaking, be placed immediately after a semicolon that ends a declaration in SDL. More precisely it is allowed to place a #CODE directive after the semicolon that ends:

- A heading of a system, block, substructure, process, procedure
- The formal parameters of a process or procedure
- The definition of a block, process, procedure
- The definition of a channel, signal route, signal, signal list, newtype, syntype, synonym, generator, connection, valid input signal set, variable, view, import, timer, remote variable, remote procedure

In the following small PR example the allowed positions are marked with an * followed by a number.

Example 383: #CODE Directive ——————————————————

```
system s; *1
  signal s1, s2(integer); *2
  channel c1 from env to b1
    with s1, s2; *3
  newtype n
    ...
  endnewtype n; *4
  block b1; *5
    signalroute sr1 from env to p1
      with s1, s2; *6
    connect c1 with sr1; *7
    process p1 (1,1); *8
      signalset s1, s2; *9
      dcl a n; *10
      start;
        ...
      state ...;
        ...
    endprocess p1; *11
  endblock b1; *12
endsystem s1;
```

——————————————————————————————————————

A code directive is considered to belong to the unit where it is defined and the declarations within the directive are thus placed among the other C declaration for that unit. In the example above, directives at positions 1, 2, 3, 4, 12 belong to system s, directives at positions 5,6,7,11 belong to block b1, while directives at positions 8, 9, 10 belong to process p1. Only one code directive may be placed at each available position.

Note:

A variable declared in a #CODE directive that belongs to a process will be shared between the process instances of the process instance set. Such a variable should only be used to represent some common property of all the process instances. To have a variable that is local to a process instance, the variable should be defined in SDL using DCL.

In the generated code the type sections are included in the order of appearance in SDL. However, the type sections are also sensitive for their relative position comparing with SDL sort definitions. This means that the order of the type definitions in the system in the example above will be as follows:

1. Type sections in 1, 2, 3
2. Type generated for newtype n
3. Type sections in 4, 12

As the Cadvanced/Cbasic SDL to C Compiler will generate the SDL sorts in the correct order, definition before usage, in C, the full algorithm is as follows.

- Step through all definitions in SDL in the order of appearance and include:
  - the type section of #CODE directives and #ADT directives.
  - generate typedef for a sort that have no reference to some other not yet generated sort.
- Step through all sorts that have not been generated, checking whether each sort references some other sort that has not been generated. If a sort does not reference some other un-generated sort, it requires typedef generation.
- Repeat the previous step until all sorts have been generated, or until no more sorts can be generated. If sorts remain not generated at this step a recursive dependency has been detected.

The heading sections are placed in the order of their appearance in SDL. This applies to the body sections as well. All body sections will be placed after the sequence of heading sections and the heading section will be placed after all the type definitions. The SDL declarations made in the corresponding unit are available in the code directives and can as usual be reached using the #SDL directive. All declarations made in

code directives are of course available in code directives in tasks in the corresponding unit or in its subunits.

The general hints on how to write C code that fits into a generated C program given in the section "Implementation of User Defined Operators" on page 2609 and in the section "Accessing SDL Names in C Code – Directive #SDL" on page 2654 are also applicable here.

Including C Code in SDL Expressions – Operator #CODE

For each sort defined in an SDL system, both predefined and user defined, the Cadvanced/Cbasic SDL to C Compiler includes an operator #CODE with the following signature:

```
#CODE : Charstring -> S;
```

where S is replaced by the sort name. This operator or rather these operators make it possible to access variables and functions defined in C using the #CODE directive in SDL expressions and still have syntactically and semantically correct SDL expressions.

During code generation, the code generator will just copy the Charstring parameter at the place of the #CODE operator.

Example 384: #CODE Directive ––––––––––––––––––––––––––––––

Suppose that `x` and `y` are SDL variables, which are translated to `z72_x` and `z73_y`, that `a` and `b` are C variables, and `f` is a C function defined in #CODE directives.

SDL expression	C expression
`x + #CODE('a')`	`z72_x + a`
`x + #CODE('a*b')`	`z72_x + a*b`
`x*#CODE('(a+b)')*y`	`z72_x*(a+b)* z73_y`
`#CODE('f(a,#SDL(x))')`	`f(a, z72_x)`

––

Within the Charstring parameter of a #CODE operator the #SDL directive is available in the same way as in other included C code. This is also shown in the last of the examples above.

As there is one #CODE operator for each sort in the system, it is sometimes necessary to qualify the operator with a sort name to make it possible for the SDL Analyzer to resolve which operator that has been used. If, for example, the question and all answers in a decision are given as applications of #CODE operators, then it is not possible to determine the type for the decision. One of the #CODE operators should then be qualified with a sort name to resolve the conflict.

Example 385: Code Directive

```
DECISION #CODE('a');
    (#CODE('1')) : TASK ...;
    (#CODE('2')) : TASK ...;
ENDDECISION;
```

In this case the sort of the decision cannot be resolved. To overcome this problem the question could be written as

```
DECISION TYPE integer #CODE('a');
```

Names and Prefixes in Generated Code

When an SDL name is translated to an identifier in C, a prefix is normally added to the name given in SDL. This prefix is used to prevent name conflicts in the generated code, as SDL has other scope rules than C and also allow different objects defined in the same scope to have the same name, if the objects are of different entity classes. It is, for example, allowed in SDL to have a sort, a variable and a procedure with the same name defined in a process. So the purpose of the prefixes is to make each translated SDL name to a unique name in the C program.

A generated name for an SDL object contains four parts in the following order:

1. The character “z”
2. A sequence of characters that make the name unique. If the object is part of a package, the package name will appear in this sequence.
3. An underscore “_”
4. The SDL name stripped from characters not allowed in C identifiers

Sequence of Characters

A C identifier may contain letters, digits, and underscore “_” and must start with a letter.

The sequence of characters that make the name unique is determined by the position of the declaration in structure of declarations in the system:

- Each declaration on a level is given a number: 0, 1, 2, .., 9, a, b, .., z.

- If the number of declaration on a level is greater than 36, the sequence is: 00, 10, 20, .., 90, a0, .., z0, 01, 11, 21, .., 91, a1, .., z1, ..., 0z, 1z, 2z, .., 9z, az, .., zz.
- If the number of declarations is greater than 36 * 36 then three character sequences are used, and so on.

The total sequence making a name unique is now constructed from the "declaration numbers" for the unit and its parents, that is the units in which it is defined, starting from the top.

If, for example, a sort is defined as the 5th declaration in a block that in turn is the 12th declaration in the system, then the total sequence will be b4 (if not more than 36 declarations are present on any of the two levels).

Example 386: Generated Names in Code ––––––––––––––––––––––

Examples of generated names:

SDL Name	Position of the Declaration	Generated Name
S1	10th declaration in the system	z9_S1
Var2	3rd declaration in the process, which is the 5th declaration in the block, which is the 15th declaration is system	ze42_Var2

––

There will also be other generated names using the prefixes. If, for example, a sort MySort is translated to za2c_MySort, then the equal function connected to this type (if it exists) will be called yEq_za2c_MySort.

Prefixes

Note:

If the OO diagram types in SDL-92 are used (system type, block type, process type), **full prefix should always be used**, as the OO concepts in itself most likely mean the name conflicts will be introduced in C.

This strategy for naming objects in the generated code should be used in all normal situations, as it guarantees that no name conflicts occur. The Cadvanced/Cbasic SDL to C Compiler offers, however, possibilities to change this strategy. In the *Make* dialog in the Organizer (see "Make" on page 119 in chapter 2, *The Organizer*) and in the user interface an Analyzer running stand-alone (see *"The Analyzer Command-Line UI" on page 2404 in chapter 55, The SDL Analyzer*), it is possible to select one of the following strategies: *full* prefix, *entity class* prefix, *no* prefix, or *special* prefix. Full prefix is default and is the strategy described above.

Entity Class Prefix

If entity class prefix is selected, then the prefix that is concatenated with the SDL name will be in accordance with the table below and depends only of the entity class of the object.

Entity class	Prefix	Entity class	Prefix
Block, block type, block instance	`blo`	Process, Process type, Process instance	`prs`
Block substructure	`bls`	Remote procedure	`rpc`
Channel	`cha`	Remote variable	`imp`
Channel substructure	`chs`	Service, Service type, Service instance	`ser`
Connection	`con`	Signal	`sig`
Formal parameter	`for`	Signal list	`sil`
Gate	`gat`	Signal route	`sir`
Generator	`gen`	Sort = Newtype	`sor`
Import	`imp`	State	`sta`
Label	`lab`	Syntype	`syt`
Literal	`lit`	Synonym	`syo`
Operator	`ope`	System, System type	`sys`
Package	`pac`	Timer	`tim`
Predef	`pre`	Variable	`var`
Procedure	`prd`	View	`vie`

Using entity class prefix means that the user must guarantee that no name conflict occurs. It also means, however, that the generated names are predictable and thus simplifies writing C code where the SDL names are used. It is only necessary to look for name conflicts within entity classes, for example not having two sorts with the same name. The entity class prefixes handle the case when two objects of different entity class have the same name. Note that the table above contains all entity classes. Not all of the items are actually used by the code generator.

No Prefix

The third alternative, no prefix, means of course that no prefixes are added to the SDL name. The name in the C program will then be the SDL name stripped form characters that are not allowed in C identifiers (everything except letters, digits, and underscore). In this case, the user must guarantee that no name conflict occurs and that the stripped name is allowed as a C identifier, that is, that it begins with a letter.

Special Prefix

In the fourth alternative, special prefix, full prefixes are used for all entity classes except variable, formal parameter, sort, and syntype. For these entity classes no prefix is used.

Conclusion

As was said in the beginning of this subsection, the user should have a good reason for selecting anything but the full prefix, as it could be very difficult to spot name conflicts. The C compiler will in some cases find a conflict, but may in other cases consider the program as legal and generate an executable program with a possibly unwanted behavior. The note above about OO concepts is also a strong argument for full prefix.

Case Sensitivity

Another aspect concerning identifiers is that SDL is case insensitive, while C is case sensitive. The Cadvanced/Cbasic SDL to C Compiler has two translation schemes for identifiers, one is to use the capitalization used in the declaration of the object of concern (default), and one is to use lower case identifiers. The translation scheme is selected in the *Make* dialog in the Organizer (see "Make" on page 119 in chapter 2, *The Organizer*) or in the user interface of the Analyzer, when it is executed stand-alone (*"The Analyzer Command-Line UI" on page 2404 in chapter 55, The SDL Analyzer*).

Specifying Names in Generated Code – Directive #NAME

If you wish to decide the name of an object in generated code yourself you can use the #NAME directive. Place the directive directly after the name in the declaration of the object. It should contain the desired name to be used in the generated code within quotes.

Example 387: #NAME Directive

```
NEWTYPE S /*#NAME 'S' */ STRUCT
  a integer;
  b Boolean;
  ADDING OPERATORS
    Op /*#NAME 'OtherName' */ :
      S, S -> Boolean;
ENDNEWTYPE;
```

The name defined in a #NAME directive will be used everywhere that the SDL name is used in the generated code, with two exceptions:

- In the monitor system the SDL names will be used in the communication with the user.
- The name of the files for generated code are not affected by the usage of #NAME directives.

There are, however, some restrictions on where #NAME directives can be placed. Some objects in, for example, a block type are generated in each instantiation of the block type. If a name directive is placed at such an object, the name will probably be overloaded in C, resulting in a C compilation error. The sensitive objects are processes, process instantiations, signal routes, channels, remote variables, and remote procedures.

Assigning Priorities – Directive #PRIO

Priorities can be assigned to processes and process instantiations using the directive #PRIO. The process priorities will affect the scheduling of processes in the ready queue, see "Time" on page 2576. A priority is a positive integer, where low value means high priority. #PRIO directives can be placed directly after the process heading in the definition of the actual process or **last in the reference symbol** (in SDL/GR).

Example 388: #PRIO Directive in process headings ––––––––––––– .

```
Process P1; /*#PRIO 3 */
Process P2(1,1); /*#PRIO 5 */

Process P3 : P3Type; /*#PRIO 3 */
Process P4(1,1) : P4Type; /*#PRIO 5 */
```

––

Processes that do not contain any priority directive will have the default priority 100.

Initialization – Directive #MAIN

The #MAIN directive is used to include initialization code that should be executed before any process transitions are started. The directive should be placed in the system definition directly after the system heading.

Example 389: #MAIN Directive––––––––––––––––––––––––––––––

```
System S;
/*#MAIN
C code for initialization */
```

––

The #MAIN directive has exactly the same structure as the #CODE directive for including code in tasks. The included code will, however, be placed last in the yInit function, after the initialization of the internal structure, but before any transitions are executed.

Modifying Outputs – Directive #EXTSIG, #ALT, #TRANSFER

The purpose of these directives is to modify the standard behavior of an SDL output. The #EXTSIG directive can be used to build applications with the SDL suite run-time library. The directives #ALT and #TRANSFER are only useful together with other real-time operating systems.

The directive #EXTSIG is used to replace the code for an SDL output with any appropriate in-line C code. This is an optimization and an alternative to the OutEnv function (see chapter 58, *Building an Application*). The #EXTSIG directive can be specified:

- Last in an output symbol (in PR just before the ‘;’).

- Just before the ‘,’ or ‘;’ ending the definition of a signal.

In the first case the #EXTSIG is valid for the signal(s) sent in the output symbol, and in the second case for all outputs of the defined signal.

Example 390: #EXTSIG Directive ––––––––––––––––––––––––––

```
signal
  Signal1          /*#EXTSIG */,
  Signal2(integer) /*#EXTSIG */;

output Signal3 To Sender /*#EXTSIG */;
```

––

For each output of a signal with a #EXTSIG directive (in either way described above) the following code is generated:

```
#ifndef EXT_SignalName
  "the normal implementation of an output"
#else
  EXT_SignalName(
    SignalName, ySigN_SignalNameWithPrefix,
    ToExpression, SignalParameters)
#endif
```

where SignalName is the name of the signal in SDL. The parameter ToExpression is a translated version of the SDL expression after TO in the output. If no TO expression is given in the output, this parameter will be xNotDefPId. The entry SignalParameters will be replaced by the list of signal parameter values given in the output.

The intention of this code is to give the user the possibility of introducing a macro with the same name as the signal, where the implementation of the output is expanded to in-line code. By just having a compilation switch which selects if this macro is visible or not, the same generated code can be used both for simulation and for highly optimized applications. An appropriate switch is probably XENV, which governs the normal way of connecting an SDL system to the environment.

Example 391: #CODE Directive ––––––––––––––––––––––––––––––

The following #CODE directive can be included in a text symbol in the system diagram (assuming a signal called SigName with one parameter).

```
/*#CODE
#TYPE
#ifdef XENV
#define EXT_SigName(Name, IdNode, ToExpr, Param1) \
   suitable macro code
#endif
*/
```

––

The other two directives, #ALT and #TRANSFER, can be used together with appropriate real-time operating systems, to have two different interpretations of an output (internal or external output for example) and to specify that a received signal should be immediately retransmitted (#TRANSFER). These kinds of features are not uncommon in real-time operating systems, and can be modeled and simulated by the Cadvanced/Cbasic SDL to C Compiler using these directives. Both these directives should be placed last in the output symbol.

The presence of an #ALT directive will be reflected in the generated code in the way described below.

- If no directive is used, the following macros will be present in generated code for sending a signal:
  - SDL_2OUTPUT: used when the receiver is known.
  - SDL_2OUTPUT_NO_TO: used when the receiver has to be calculated during runtime.
  - SDL_2OUTPUT_COMPUTED_TO: used when the receiver is calculated during code generation.
- If an #ALT directive is given these macros are replaced by:
  - SDL_ALT2OUTPUT
  - SDL_ALT2OUTPUT_NO_TO
  - SDL_ALT2OUTPUT_COMPUTED_TO

In the Master Library, the macros with and without ALT are expanded identically. In an OS integration they might be handled differently to implement two classes of signal sending.

The presence of a #TRANSFER directive indicates that a signal should be directly retransmitted to some other receiver. This can of course be

performed in SDL by an input-output, but then it is necessary to create a new signal and copy the contents of the received signal to the new signal. Using the #TRANSFER directive this copying can be avoided.

In generated code the macros

- ALLOC_SIGNAL
- ALLOC_SIGNAL_PAR

are generated to allocate the data area for a new signal. If a #TRANSFER directive is present in the output statement, these macros are replaced by:

- TRANSFER_SIGNAL
- TRANSFER_SIGNAL_PAR

Note:

In the master library #TRANSFER will still be implemented as a signal copy. It may be possible in an OS integration to avoid the copying if the OS supports such actions.

Normally the #TRANSFER directive should be used in the following way:

```
INPUT signal1(,,,);
OUTPUT signal1(,,,) /*#TRANSFER*/;
```

That is, receive none of the signal parameters in the input and retransmit the signal unchanged. If you want to receive, for example, the second parameter (in variable Var1) and retransmit the signal unchanged except for parameter 3, that should have a new value (73), the following code can be used:

```
INPUT signal1(,Var1,,);
OUTPUT signal1(,,73,) /*#TRANSFER*/;
```

Linking with Other Object Files – Directive #WITH

Note:

This feature is only valid for SDL/PR as input, when the Analyzer is executed stand-alone. Similar features are available in the Organizer's *Make* dialog among the *Generate makefile* options, see "Generate makefile" on page 122 in chapter 2, *The Organizer*.

You can tell the Cadvanced/Cbasic SDL to C Compiler that a number of user defined and precompiled units should be linked together with a generated simulation program. You do this in a #WITH directive that should be placed in the system definition directly after the system heading.

Example 392: #WITH Directive

```
System S; /*#WITH 'file1.o' 'file2.o' */
```

Within the #WITH directive the object files that are to be part in the link operation should be given between quotes, as in the example above. These files will be included in the definition of the link operation in the generated `.m` file.

The make file will, however, not include any definition of how to compile the corresponding source files, as it is impossible for the code generator to know the compilation options or even what compiler the user wants. A user that knows how to interface routines in other languages in a C program, can with this knowledge and the #WITH directive link modules written in another language together with the generated program.

Note:

The #WITH directive will only affect the generated make file. There will be no change in the generated C code when a #WITH directive is introduced.

Naming Tasks in Trace Output – Directive #ID

To simplify the identification of a TASK in a trace printout, the Cadvanced/Cbasic SDL to C Compiler uses the variable on the left hand side of the first assignment statement or the first informal text in the task symbol. A user that is not satisfied with this can name the tasks using #ID directives. An ID directive should contain the character string that is to be used as identification in trace printouts.

Example 393: #ID Directive ––––––––––––––––––––––––––––––––––

```
/*#ID 'Identification of task' */
```

––

The code generator will use the first ID directive it finds in a TASK (if any). An ID directive may be placed:

- First in the task (in PR that is just after the keyword TASK)
- Immediately before or just after a comma separating two assignments or two informal texts
- Last in a task (in PR that is just before the semicolon)
- Directly after the semicolon (this position is only available in SDL/PR).

Directive #C, #SYNT, #SYNTNN, #ASN1

These directives are used to pass information from the tools generating SDL from other languages, for example for C (a `.h` file) or ASN.1.

The #C and the #ASN.1 directive will be inserted after the semicolon ending the package definition or after the package name.

```
package asn1_module; /*#ASN.1 'Module_Name' */
package asn1_module /*#ASN.1 'Module_Name' */;
```

The #ASN.1 directive contains the ASN.1 module name and the #C directive contains the name of the originating `.h` file.

The #SYNT directive and its special form the #SYNTNN directive are used in a package generated from C, to indicate which SDL sorts that are synthesized, i.e. which sorts that where needed in SDL but do not have a name and definition in C. As the originating `.h` file is included in the generated code (with a #include), no typedefs should be generated for the non-#SYNT sorts, while typedefs have to be generated for the synthesized sorts. The #SYNT directive is inserted directly after the name of the newtype or syntype.

Alternative Implementations of the String Generator – Directive #STRING

An instantiation of the string generator can be translated to either a linked list or an array when implemented in C. The #STRING directive is used to determine which translation method to select. The directive should be inserted in a NEWTYPE definition, directly after the string name, when defining a new string data type.

If the #STRING directive is not included in the NEWTYPE definition, the string is implemented as a linked list. This is the default translation method. However, just adding the directive to the NEWTYPE definition is not enough to implement the string as an array. The length of the array must also be defined. This can be done either by using a directive parameter or by using a size constraint in the NEWTYPE definition of the string generator. How this is done is presented in the examples below.

Linked List Implementations

The following examples show how to translate an instantiation of the string generator to a linked list.

Example 394: No #STRING Directive

```
NEWTYPE Example_String
    string(integer, empty)
    constants size (0:10)
ENDNEWTYPE;
```

If the #STRING directive is missing from the NEWTYPE definition, the string will be implemented as a linked list. The length of the list is unlimited, unless a size constraints is defined. In this case the length of the string is within the range of 0 and 10.

Example 395: #STRING Directive without Limited String Size

```
NEWTYPE Example_String /*#STRING */
    string(integer, empty)
ENDNEWTYPE;
```

In this example the string is also implemented as a linked list. The reason for this is that we have not defined a maximum length of the string.

Example 396: #STRING Directive with Parameter Value 0 ––––––––––

```
NEWTYPE Example_String /*#STRING 0*/
    string(integer, empty)
    constants size (0:10)
ENDNEWTYPE;
```

If the parameter value of the #STRING directive is 0, the directive is ignored. Therefore the string will be implemented as a linked list. The size constant will still decide the length of the string.

This example, however, shows how to gradually migrate from a linked list implementation to an array implementation. By just changing the parameter value to anything larger than 0, this NEWTYPE definition creates an array implementation.

––

Array Implementations

The following examples show how to translate an instantiation of the string generator to an array.

Example 397: #STRING Directive with Limited String Size ––––––––

```
NEWTYPE Example_String /*#STRING */
    string(integer, empty)
    constants size (0:10)
ENDNEWTYPE;
```

In this case the string is implemented as an array with the maximum length of 10.

––

Example 398: #STRING Directive with Parameter Value ––––––––––

```
NEWTYPE Example_String /*#STRING 100*/
    string(integer, empty)
    constants size (0:10)
ENDNEWTYPE;
```

In this example the string will be implemented as an array with the length of 100. The parameter value in the #STRING directive overrides the size constant, which in this case is redundant.

––

Size constraint

The size constraint is decided by adding a constant in the NEWTYPE definition. The following declarations are valid when defining a maximum size of the string:

```
constants size (a:b)
constants size (a)
constants size (=a)
constants size (<a)
constants size (<=a)
```

Differences between the Implementations Methods

The different implementation methods affect the behavior and performance of the generated code. The following general statements apply to the methods:

- The performance of the array implementation is usually better than that of the linked list implementation.
- If the number of string values are equal to or close to the maximum string length, the array implementation is smaller.
- If the number of string values are substantially smaller than the maximum string length, the linked list requires less memory.
- The linked list requires memory allocation, while the array does not.

Selecting implementation Methods

It is of course hard to advice which method to select, but the following recommendations apply:

- If the string size is not known, use the linked list.
- If the number of string values is substantially smaller than the maximum string length, use the linked list if speed is not very important.
- If the maximum length is not large or the number of string values are almost equal to the maximum length, use the array.

Using Cadvanced/Cbasic SDL to C Compiler to Generate C++

General

The C code in the Master Library and the C code generated by the Cadvanced/Cbasic SDL to C Compiler is in the common subset of C and C++, and will thus compile both as a C program and as a C++ program. There is one special feature in the code generator concerning C++ when it comes to abstract data types and the possibility to match a C++ class and an SDL data type. Otherwise all the features for including C code, directive #ADT, directive #CODE (see "Abstract Data Types" on page 2586 and "Accessing SDL Names in C Code – Directive #SDL" on page 2654), and so on, are directly applicable for C++ as well. The #CODE directives make it possible to include class definitions as C++ code and the utilization of the classes as C++ code in SDL tasks.

Example 399: Using C++ Classes ––––––––––––––––––––––––––––

CODE directive containing declarations (see "Including C Declarations – Directive #CODE" on page 2658) which should be placed among the SDL declarations:

```
/*#CODE
#HEADING
class TEST {
public:
  void putvar(int avar, int bvar)
    {a = avar; b = bvar;}
  int  geta()
    {return a;}
  int  getb()
    {return b;}
private:
  int a,b;
} TESTvar;
*/
```

Example of usage of the class in a CODE directive in a TASK (see "Including C Code in Task – Directive #CODE" on page 2656).

```
TASK '' /*#CODE
          TESTvar.putvar(#(I), #(I)+10); */;
```

Example of usage of the class in a CODE operators in expressions (see "Including C Code in SDL Expressions – Operator #CODE" on page 2662).

```
OUTPUT Score(
  #CODE('TESTvar.geta()'),
  #CODE('TESTvar.getb()'),
  #CODE('#(TClass)->getVar()')
);
```

Connection Between C++ Classes and SDL

To obtain a close connection between a C++ class and SDL, an abstract data type in SDL can be used; see below.

If you have a C++ class (from a class library or developed specifically for the project), then the following correspondence rules can be used to map the class on an abstract data type.

C++	SDL
Class definition	Abstract data type definition
Class instance pointer	Process variable
Member functions	Operators
new, delete	Operators

Example 400: SDL and C++ Class

Suppose we have a C++ class with the following interface (`.h` file):

```
class TestClass {
public :
  TestClass (int);
  TestClass ();
  ~TestClass ();
  int updateVar(int);
  int getVar();
private :
  int v;
};
```

then the following abstract data type can be used to represent the class (in the example the NAME directive, see "Specifying Names in Generated Code – Directive #NAME" on page 2667, is used to instruct the Cadvanced/Cbasic SDL to C Compiler which name to use in C for particular SDL objects):

```
NEWTYPE TestClass     /*#NAME 'TestClassPtr' */

  LITERALS
```

```
    newTestClass      /*#NAME 'new1TestClass' */
    ;

  OPERATORS
    newTestClass      /*#NAME 'new2TestClass' */
      : integer -> TestClass;

    deleteTestClass   /*#NAME 'deleteTestClass' */
      : TestClass -> TestClass;

    updateVar         /*#NAME 'updateVar' */
      : TestClass, integer -> integer;  /*#OP(HC) */

    getVar            /*#NAME 'getVar' */
      : TestClass -> integer;           /*#OP(HC) */

/*#ADT(T A(S) E(S) D(H) H P)

#TYPE
#include "TestClass.h"
typedef TestClass * TestClassPtr;
COMMENT((NOTE! SDL data type TestClass is
        pointer to C++ class TestClass))

#HEADING
#define yDef_TestClassPtr(p)  *(p) = 0
#define new1TestClass()   new TestClass()
#define new2TestClass(P)  new TestClass(P)

extern TestClassPtr deleteTestClass
  (TestClassPtr);

#BODY
extern TestClassPtr deleteTestClass
  (TestClassPtr P)
{
  delete P;
  return (TestClassPtr)0;
}
*/
ENDNEWTYPE;
```

Note that the #ADT specification means that no code will be generated for the abstract data type. The abstract data type can utilized in the following way:

```
DCL
  TClass TestClass,
  I      Integer;

TASK TClass := newTestClass;
TASK TClass := newTestClass(2);
TASK I := updateVar(TClass, I);
TASK I := getVar(TClass);
```

```
TASK TClass := deleteTestClass(TClass);
```

The only feature that is not described before is the C option in the #OP directive. C (class) is an alternative to I (infix) and P (prefix), and specifies that the operator call should be translated to a member function call of a C++ member function. #OP(C) means that an operator call

```
F(a, b, c)
```

is translated to

```
a->F(b, c)
```

Note:

This means that each operator mapped on class member function should have the class instance pointer as first parameter.

Restrictions

SDL Restrictions

The Cadvanced/Cbasic SDL to C Compiler handles the majority of SDL concepts according to the definition of SDL-92. There are however a number of restrictions that are discussed in this section.

Analyzer Restrictions

The restrictions in the SDL Analyzer are, of course, also valid in the Cadvanced/Cbasic SDL to C Compiler. For more information see "SDL Analyzer" on page 37 in chapter 2, *Release Notes, in the Release Guide*.

Cadvanced/Cbasic SDL to C Compiler Restrictions

The Cadvanced/Cbasic SDL to C Compiler introduces more severe restrictions on the allowed set of SDL concepts than the Analyzer. For more information, see "SDL to C Compiler" on page 40 in chapter 2, *Release Notes, in the Release Guide*.

Migration Guide for Generic Functions

General

This section provides help to migrate a system using old-style code generation for operators to the new Generic Function style.

Introduction

The basic idea of the generic function approach is to decrease the number of generated help functions and functions for operators in predefined generators. This is accomplished by generating generic functions that can be re-used by different types.

This means, for instance, that only one assignment function is created. This function, however, can be used by all types. In the old-style method, one assignment function was created for each type. For operators in predefined generators, there is now **one single** length function calculating the length of **all** string generator instantiations.

In order to implement the generic functions, the parameter passing mechanisms have been changed. In principle, a generic function cannot take a value as parameter, it must receive a pointer to the value. This approach has a positive effect on the performance. However, the generic function approach introduces incompatibility problems if your existing system calls generated functions from inline C-code. If this is the case, the function calls must be changed.

If you need to migrate an SDL system created with the old-style code generation, you must solve the incompatibility issue.

References to Information

Generic Functions

The major source of information regarding generic functions is "Abstract Data Types" on page 2586, which contains a number of sections discussing different aspects of data types and operators:

- "Translation of Sorts" on page 2595 describes how different SDL types are translated to C.

- "Parameter Passing to Operators" on page 2606 discusses the general parameter passing principles for operators and literal functions. This section also lists which types that are passed as values and which types that are passed as addresses.
- "Implementation of User Defined Operators" on page 2609 describes how to include your own implementation in C for an operator.
- "Generic Functions" on page 2623 introduces you to the type info node concept and describes the general operators assign, equal, free, etc.
- "Generic Function for Operators in Pre-defined Generators" on page 2630 describes the operators in the predefined generators.
- "More about Abstract Data Types" on page 2634 comprises information on how to change the implementation of a data type.

SDL Data Types

Information on SDL data types and operators, seen from the SDL point of view, is available in "Using SDL Data Types" on page 42 in chapter 2, *Data Types*, *in the SDL Suite Methodology Guidelines*. Although this section does not discuss the use of generic functions, it provides the framework for SDL data types and operators. In some circumstances it might be better to rewrite a data type or operator in SDL, than to fix the problems in C. A number of extensions and improvements have been included in the support for data types.

Type Info Nodes

For more information on the contents of the type info nodes, please see "Type Info Nodes" on page 2979 in chapter 62, *The Master Library*, *in the User's Manual*. This section does not cover migration aspects, but provides implementation details for an interested reader.

Migrating Strategy

Overview

The common problems that might occur when migrating a system from the old-style operator implementation to the generic functions implementation can be divided into two groups:

- Some user-defined SDL operators have changed their prototypes in C. This means that user-provided implementations of the operators have to be changed and that calls to such operators directly from C have to be changed.
- Predefined operators like assignment, equal test, make, and so on, as well as operators in predefined generators might have changed their prototypes in C, which means that calls to such operators directly in C have to be changed. Also types where the implementation of assignment, equal and similar operators are changed by a #ADT directive, might have to be updated.

The first of these problems is fairly straight forward to fix, while the second might be more complex.

Step 1: Identifying Migration Problems

Before continuing with the migration instructions, check if your SDL systems are affected by any incompatibility problems.

1. Find an SDL system that compiles without errors in a previous version of the SDL suite (with the old-style operator implementations)
2. Open it in the 4.5 version of Telelogic Tau.
3. Compile it and see if you get any compilations errors.

If your compilation errors are similar to the errors in Example 401 on page 2684 you can assume that you have a migration problem.

Example 401: Compilation Errors ––––––––––––––––––––––––––

The compilation error in this example originates from the GNU compiler, gcc. It gives a fairly good feeling of what kind of errors you can expect.

```
file.c:50825: conflicting types for 'yAss_example'
file.c:50844: conflicting types for 'yEq_example'
file.c:51496: conflicting types for 'op1'
file.c:211376: too many arguments to function 'op1'
file.c:211376: cannot convert to a pointer type
file.c:6042: incompatible type for argument 2 of
'GenericAssignSort'
file.c:6081: incompatible type for argument 2 of
'op1'
```

––

Step 2: Locating the Compatibility Problems

Note:

Before you try to investigate and correct any error from the list of C compilation errors, you should perform this step. The reason is that many of the errors will point at the wrong place so correcting them might introduce errors rather than correcting anything.

The Cadvanced, Cbasic, and Cmicro SDL to C compilers allows you to find the operators and literals that should be updated. By setting the environment variable `SDT_COMP_WARN` the code generators will produce a file called `compatibility.warn` in the target directory, while generating code for the system. In the first section of this file all operators and literals that might have to be changed are listed.

Example 402: Contents of `compatibility.warn`

```
LITERAL NewDb  C-name: z0V0_NewDb
  <<SYSTEM accesscontrolooa>>
  #SDTREF(TEXT,file.sdl,57,12)
  Literal function result passed as address

OPERATOR ValidateCard  C-name: z0V1_ValidateCard
  <<SYSTEM accesscontrolooa/TYPE CardDbType>>
  #SDTREF(TEXT,file.sdl,59,5)
  Parameter 2 passed as address
    Used in DIRECTIVE at #SDTREF(TEXT,file.sdl,62)
    Used in DIRECTIVE at #SDTREF(TEXT,file.sdl,62)
```

This example shows one literal and one operator that have to be updated:

- The first line contains the name of the operator/literal in SDL and in C.
- The second line contains an appropriate qualifier, that gives information on where in the system the item is defined.
- The third line is the SDT reference to the operator/literal. This can be used in the Organizer's *Goto Source* feature to show where the SDL source is located.
- The following lines indicate where changes are needed. Each line states that the result or a parameter is passed as an address. This means that previously this item was passed as a value, but now it

should be passed as an address. A complete list of types that must be changed can be found in "Mapping Table" on page 2588.

- Last, a number of cross references might be found. These show places where the operator is called from inline C using an #SDL directive. These calls might have to be updated.

Step 3: Updating Operators and Literals

For each operator or literal that is listed in the file `compatibility.warn`, perform the instructions presented in this section.

> **Note:**
> Literals should be treated as operators without parameters.

Updating the Headers

The headers of the corresponding C functions must be updated. If you have specified #OP(B), the header is generated and thus already correct, but if you have specified #OP(H), you have included the header in C probably in the #HEADING section in the #ADT directive for the type.

1. For each parameter that should be passed as an address, add a '*' after the corresponding type name in C.
2. If the result should be passed as an address, add a '*' after the result type in C, and add an extra parameter with the same C type as the updated function result, last among the parameters.

When this step has been performed for all types, the C compilation errors will be reliable again.

Example 403: Headers ----------

```
extern str lit1 (void);
extern str op1 (str, SDL_Integer);
```

Assume `str` is the type that should be passed as an address. The results of both functions and the first parameter of `op1` are mentioned in the `compatibility.warn` file. The heading should be updated to:

```
extern str* lit1 (str*);
extern str* op1 (str*, SDL_Integer, str*);
```

Updating Parameter Specifications

The parameter specification and result in the function implementation must be updated to match the heading. The function implementation is probably in the #BODY section in the #ADT directive of the type.

Example 404: Parameter Specifications (continued from Example 403)

```
str lit1 (void)
{ .... }
str op1 (str P1, SDL_Integer P2)
{ .... }
```

should be changed to:

```
str* lit1 (str* Result)
{ .... }
str* op1 (str* P1, SDL_Integer P2, str* Result);
{ .... }
```

Updating Operator/Literal Functions

The implementation of the operator/literal functions must be updated to reflect the change in parameters.

Example 405: Operator/Literal Functions (continued from Example 404) ----------

In the function `op1` every occurrence of:

`P1` should be replaced by `(*P1)`

Special cases where other changes might be more appropriate:

`&P1` should be replaced by `P1`
`P1.abc` should be replaced by `P1->abc`

The new Result parameter must be assigned the result value of the function and the function must end with a return statement.

Example 406: Return Parameter ----------

```
return Result;
```

It is not unusual that the function contains a local variable used for calculating the result. Normally this variable is no longer needed.

While updating the implementation of the function, information from the next section on, for example, assign and equal function might be valuable.

Note:

Check that parameters that are passed as addresses are NOT CHANGED within the function. If that is the case, copy the value to a local variable first, and work with that variable.

Update Calls to Functions

The calls to the changed functions must be updated. Calls made from SDL cause no problems as the SDL compilers produce the correct code. Only calls made directly from inline C code may have to be updated. The operator list in the file `compatibility.warn` contains cross-references to the inline C code where the operator is called using an `#SDL` directive.

Note:

Operator calls in C without the `#SDL` directive, are not listed in `compatibility.warn`. These calls can only be found via the compiler error list.

For parameters that has changed parameter passing mechanism from pass as value to pass as address you should perform one of the following tasks:

- If the actual parameter is a variable (or something it is possible to take the address of), add a '&', before the variable.
- If the actual parameter is a call to a function, then probably this function has changed its prototype so that it now returns an address. In that case nothing needs to be performed. In other cases proceed to the next possibility.
- If none of the situation above is appropriate, insert a new function local variable of the parameter type, assign the value of the actual parameter to this variable, and insert the address of the variable as actual parameter.

Implementation Hint

> **Note:**
>
> The following information is not required for the migration of the system, but can be used to improve the performance of the system.

When updating the operators, it might be worth investigating the available features in the SDL suite, including extensions for operators, in/out parameters, no parameters, no result, etc.

One not too uncommon situation is when a value is passed as an in parameter, then changed by the operator and returned as result value. In every operator call, the same variable is used as both the actual parameter and the receiver of the result. To improve speed of the application, the operator, could be changed to an operator without result and using in/out parameters.

If you perform a change like this, remember to use the cross-reference tool to find all places where the operator is used.

Step 4: Updating typedefs

Overview

Another area where backward incompatibility problems might be present is when the `typedef` is changed in an #ADT directive, especially if the assign, equal, or free functions are changed as well. In the second section of the file `compatibility.warn` all types changing the typedef and at least one of assign, equal, or free functions are listed.

Example 407: Typedef Change ––––––––––––––––––––––––––––––

```
NEWTYPE example  C-name:zDZ_example
  <<SYSTEM mysystem>>
  #SDTREF(TEXT,file.sdl,3096,9)
    Pass as Value  #ADT(T(B)A(B)E(B)F(B))
```

––

The example should be interpreted like this:

- The first line contains the name of the newtype in SDL and in C.
- The second line contains an appropriate qualifier that gives information on where in the system the newtype is defined.

- The third line is the SDT reference to the newtype. This can be used in the Organizer's "goto source" feature to locate the SDL source code.
- The fourth line contains either "Pass as Value" or "Pass as Address", depending on the property of the newtype, followed by the interpretation of the #ADT directive.

As the #ADT directive can be used in many different ways, it is impossible to describe a general method how to correct any problems. The type list in the `compatibility.warn` file indicates what newtypes that have highest probability to cause problems. It is recommended that you go through the listed newtypes and review them given the information in the following sections.

You also have to find and in some cases correct the places where the functions discussed below are called. The compiler error list is the main source of information for this task. Please see "Locating Source Code" on page 2693 if you have problems locating the corresponding SDL source listed in a C compilation error message.

Assignment Functions

For all types passed as addresses, the differences between the old-style, and the new generic assignment functions are:

- The old functions pass the value of the right-hand side expression as the second parameter, while the new pass the address of the right-hand side expression.
- The old functions returns void, but the generic functions return the first parameter, i.e. the address of variable.
- It is now required that the assignment function must be a function (it cannot be a macro), as the address of the function is stored in the type info node for the newtype.

Example 408: The Assignment Function ––––––––––––––––––––––

The prototype for an old assign function would be:

```
void yAss_typename (typename*, typename, int)
```

The prototype for a new assign function should be:

```
typename* yAss_typename (typename*, typename*, int)
```

The body of the assignment function must be updated for these changes.

If assignment for a type passed as an address is used in inline C code, one of the following tasks must be performed:

- If the expression parameter is a variable, a '&', should be added before the variable.
- If the expression is an SDL operator call, a change is normally not needed, as the operator in the generic function model will return the address of a value, not the value itself (for types passed as address).
- You might want to add (void) before the yAss call to tell the compiler that you want to ignore the result.

For more information please see "Generic Assignment Functions" on page 2624.

Equal Functions

For all types passed as addresses, the differences between the old and the new generic equal functions are:

- The old functions pass the values of the two expression, while the new generic equal functions pass the addresses of the expressions.
- It is now required that the equal function must be a function (it cannot be a macro), as the address of the function is stored in the type info node for the newtype.

Example 409: The Equal Function ----------

The prototype for an old equal function would be:

```
SDL_Boolean yEq_typename (typename, typename)
```

The prototype for a new equal function should be:

```
SDL_Boolean yEq_typename (typename*, typename*)
```

The body of the equal function must be updated for these changes.

If equal for a type passed as an address is used in inline C code, perform the same tasks as presented for the assignment function.

For more information please see "Generic Equal Functions" on page 2627.

Free Functions

The `yFree_typename` function is backward compatible. However, it is now required that the equal function must be a function (it cannot be a macro), as the address of the function is stored in the type info node for the newtype.

For more information please see "Generic Free Functions" on page 2628.

Default Functions

The `yDef_typename` macro or function from the non-generic mode is no longer used or generated. Initialization of SDL variables are performed as follows:

- if the variable has an initial value given in the declaration, the variable is assigned this value, using an assignment statement.
- else if the type of the variable has a default value, the variables are assigned this value, using an assignment statement.
- else the variable is set to 0 using memset. If 0 is not an appropriate initial value, the function GenericDefault is called to initialize the variable. Examples of types where 0 is not an appropriate value are structs with components with initial values, Object_identifiers, Strings, general Powersets, Bags and general Arrays.

As yDef functions/macros do not exist in Generic mode, all usage in inline C code must be changed. There are several possibilities. First, decide if assigning a default value is really necessary. If it is necessary, then some of the following principle solutions might be used:

- memset to 0
- ```
  GenericDefault(&variable,
                 (tSDLTypeInfo *)&ySDL_typename)
  ```
- ```
  yAss_typename(variable, expression,
                XASS_MR_ASS_NF)
  ```
- direct C assignment, if possible

Make Functions

The `yMake_typename` function from the non-generic mode is no longer used or generated. All calls in inline code to yMake functions must be replaced by calls to proper GenericMake functions as described in "Generic Make Functions" on page 2629.

> **Note:**
>
> yMake functions pass component values as values, while GenericMake functions pass component values as addresses

Operators in Predefined Generators

The operators in predefined generators have got new generic implementations. All the available generic implementations are listed in "Generic Function for Operators in Pre-defined Generators" on page 2630. Macros that support the old operators names and translate them to the generic functions are generated. If you get compilation errors in such a call you should compare the call with the macro and the generic function, to see if there are any differences in the parameter passing principle.

Locating Source Code

A typical error message from a C compiler lists a file name, a line number, and a description of the error. To locate the corresponding SDL source code perform the following:

1. In a text editor, open the file that is listed in the error message
2. Go to the given line number.
3. Search upwards for an SDT reference, that is, a C comment starting with:

   ```
   /*#SDTREF(
   ```

Copy the SDT reference and paste it into the *Goto Source* feature in the Organizer. The Organizer will show you where the C code originated from.

Chapter 57 The Cadvanced/Cbasic SDL to C Compiler