Chapter

66

The Cmicro SDL to C Compiler

The Cmicro SDL to C Compiler translates your SDL system into a C program that you can compile together with the Cmicro Library and the SDL Target Tester target library. The Cmicro Library and the SDL Target Tester target library is not available as a pre-linked library but is delivered as source to enable scaling of the kernel. The scaling is dependent upon the SDL system characteristics. This chapter is a reference manual for the Cmicro SDL to C Compiler.

- **In chapter 67, *The Cmicro Library*, you will find information about how to customize your own libraries for 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. The Cmicro Library is only of use when compiled with the code which is generated by the Cmicro SDL to C Compiler.**
- **In chapter 68, *The SDL Target Tester*, you will find a reference to the features, which enables testing in a host-target environment. The SDL Target Tester is applicable for the Cmicro SDL to C Compiler and the Cmicro Library only.**

Application Area for the Cmicro SDL to C Compiler

The application area for the Cmicro SDL to C Compiler is:

- Generation of applications, including embedded system applications with real time characteristics (Configuration: Cmicro Library and generated C code running on target).
- Generation of target debug applications, including embedded system applications with real time characteristics (Configuration: Cmicro Library, generated C code and SDL Target Tester running on target).

In this part of the chapter, the general behavior of the Cmicro SDL to C Compiler, as seen from the users point of view, is discussed.

Highly Optimized Code for Target

The generated code in combination with the Cmicro Library is highly optimized, which is unavoidable for microcontrollers and real-time applications. Some optimizations have been possible only by introducing restrictions in the use of SDL. Other optimizations have been possible by generating more compact code. For the restrictions in the use of SDL please see "SDL Restrictions" on page 3358. Details regarding the output of the Cmicro code generation can be found in "Output of Code Generation" on page 3326.

Target Debug

With the generated code it is possible to debug the application on the target using the Cmicro Library and the SDL Target Tester library. The parts of the Cmicro code generation which are used for the SDL Target Tester are also highly optimized. Please see chapter 68, *The SDL Target Tester*.

Overview of the Cmicro SDL to C Compiler

The SDL Analyzer, which can be invoked from the Organizer, contains an SDL parser, an SDL semantic checker, and – among other code generators – the Cmicro SDL to C Compiler.

Many options can be chosen from the user which affect the analysis of the SDL system. Furthermore, a lot of error checks are performed automatically before code generation starts. This makes it possible to improve written SDL specifications before any run-time testing must be done.

The options that the user may choose for analysis and the error checks that are performed by the analyzer are described in chapter 55, *The SDL Analyzer*.

At some places the Cmicro SDL to C Compiler can be used in exactly the same way as the Cadvanced/Cbasic SDL to C Compiler can be used. At some other places the use of this C Code Generator, or what this C Code Generator produce, is different.

The Cmicro SDL to C Compiler generally can process the same input as the Cadvanced/Cbasic SDL to C Compiler can. The differences are explained within this chapter.

The differences in the output of the both code generators are described within the subsection "Output of Code Generation" on page 3326.

The overall differences of the both code generators are described in the section "Differences between Cmicro and Cadvanced" on page 3364 in chapter 67, *The Cmicro Library*.

The following subsections describe how the Cmicro SDL to C Compiler might be used.

Generated Files

The C files, which are generated by the Cmicro SDL to C Compiler, can only be used in connection with the Cmicro Library and the SDL Target Tester. It is not possible to validate and simulate the SDL system with the C code generated by Cmicro as this code is only suitable for target applications. To simulate and validate the SDL system within the SDL suite, the user has to choose the Cbasic SDL to C Compiler. In order to view the process of generating C applications see the Organizer's *Make* dialog in chapter 2, *The Organizer*.

The SDL Analyzer, which contains the Cmicro SDL to C Compiler can also be started as a stand-alone tool. For more information about this possibility please see *chapter 55, The SDL Analyzer*.

There are several steps that must be carried out before the generated C files can be compiled and linked together with the Cmicro Library. The user should follow the procedures that are documented in the section "Targeting using the Cmicro Package" on page 3389 in chapter 67, *The Cmicro Library*.

In the following subsections the different files that are generated are explained.

Generated Configuration File

The first file that is generated from the Cmicro SDL to C Compiler is called `sdl_cfg.h`. It is used to scale the Cmicro Kernel depending on what characteristics the SDL system has. This is called automatic scaling and automatic dimensioning facility.

The file contains a header, process ID declarations, and then a `#define` or a `/*NOT define ... */` for each of the flags that the Cmicro SDL to C Compiler can generate automatically.

Example 540: The Header of an sdl_cfg.h ----------

```
/* Program generated by SDL suite.Cmicro <version> <date> */

#ifndef XSCT_CMICRO
#define XSCT_CMICRO
#endif

/* "sdl_cfg.h" file generated for system <systemname> */

#define XMK_CFG_TIME <GenerationTime>
```

The XMK_CFG_TIME macro is used internally when compiling and executing with the SDL Target Tester takes place. With this macro, a rough consistency check for the generated files is done. The <GenerationTime> of the different files that are generated is compared in the Library and by the SDL Target Tester. If there is an inconsistency, compilation errors will occur.

The rest of the file sdl_cfg.h is about the automatic scaling and automatic dimensioning of the SDL system. It may look for example like:

Example 541: The Tail of an sdl_cfg.h ----------

```
#define MAX_SDL_PROCESS_TYPES <N>
#define XMK_USED_ONLY_X_1
#define MAX_SDL_TIMER_TYPES <X>
#define MAX_SDL_TIMER_INSTS <Z>
#define XMK_HIGHEST_SIGNAL_NR 4
/* NOT #define XMK_USED_TIMER */
/* NOT #define XMK_USED_DYNAMIC_CREATE */
/* NOT #define XMK_USED_DYNAMIC_STOP */
/* NOT #define XMK_USED_SAVE */
#define XMK_USED_SIGNAL_WITH_PARAMS
/* NOT #define XMK_USED_TIMER_WITH_PARAMS */
/* NOT #define XMK_USED_SENDER */
/* NOT #define XMK_USED_OFFSPRING */
/* NOT #define XMK_USED_PARENT */
/* NOT #define XMK_USED_SELF */
/* NOT #define XMK_USED_PWOS */
/* NOT #define XMK_USED_INITFUNC */
```

For a first rough understanding of the meaning of the different flags: The SDL system from above contains <N> process types (using SDL'88 terminology), all the processes are declared in the form (0,1) or (1,1). There are <X> timers declared (in this case, <X> must be 0, because XMK_USED_TIMER is undefined, and the system uses an amount of <Z> signals. The system does not use any create or stop (XMK_USED_DYNAMIC_CREATE and XMK_USED_DYNAMIC_STOP are undefined). In this way all the other flags have special meaning.

For explanations about the different flags the user should refer to "Automatic Scaling Included in Cmicro" on page 3426 in chapter 67, *The Cmicro Library*.

Generated C File

Assumed, that the user selected *"No separation"* in the Organizer's Make Dialog, and no partitioning is used, then the Cmicro SDL to C Compiler will generate one C file per SDL system. This file contains all the characteristics of the SDL system including all the declarations that

the SDL system itself needs. For an explanation of this file see "Output of Code Generation" on page 3326.

Generated Environment Header File

There is one file generated from the Cmicro SDL to C Compiler that contains all the definitions and declarations that are necessary to implement the environment functions `xInEnv` and `xOutEnv`.

The file is generated only if the option *Environment header file* in the Targeting Experts is switched on.

The file is called `<systemname>.ifc` and it contains a header, the type definitions used on system level (newtypes, syntypes, synonyms), the signal IDs and the structure type definitions for the parameters of the signals.

Example 542: The Header of an <systemname>.ifc file

```
#ifndef X_IFC_z_env01
#define X_IFC_z_env01
#define XMK_IFC_TIME <GenerationTime>
```

The `XMK_IFC_TIME` macro is used internally when compiling and executing with the SDL Target Tester takes place. With this macro, a rough consistency check for the generated files is done. The `<GenerationTime>` of the different files that are generated is compared in the Library and by the SDL Target Tester. If there is an inconsistency, compilation errors will occur.

Caution!

As there are defines generated that contain no prefixes, there might be compiler warnings like `Illegal redefinition of macro`. Such redefinitions should never be ignored because fatal errors during run-time may occur. The user should introduce a prefix for signals or sorts with different meaning on SDL level, in order to map these names to unambiguous identifiers in C.

More explanation about the environment header file is given in chapter 67, *The Cmicro Library*.

Sorts

Followed by the header the section about sorts follows. The sorts are generated according to the documentation in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*.

Signal IDs and Parameter Structures

Next, the definitions for signals follows, which consists of:

- A C comment with a comment explaining if the signal is `IN` or `OUT` as seen from SDL
- An optional declaration of a C structure type definition (if the signal carries parameters)
- The definition of the signal ID

For easy interpretation: For an SDL signal **without** parameters, going from the environment to SDL, like:

```
signal SIn;
```

the following is generated into the `.ifc` file:

```
/* SIn IN */
#define SIn  <X>
```

For an SDL signal **with** parameters, going from SDL to the environment, like

```
signal SOut (integer, mystruct, boolean);
```

the following is generated into the `.ifc` file:

```
/* SOut OUT */
typedef struct {
    SIGNAL_VARS
    SDL_Integer  Param1;
    mystruct     Param2;
              /* mystruct declared in the section */
              /* declaring sorts                  */
    SDL_Boolean  Param3;
} yPDef_<UniquePrefix>_SOut;

typedef yPDef_<UniquePrefix>_SOut  *yPDP_<UniquePrefix>_SOut;
#define yPDP_SOut                  yPDP_<UniquePrefix>_SOut
#define yPDef_SOut                  yPDef_<UniquePrefix>_SOut
#define SOut                       <X>
```

At least the signal ID (here: `SOut`) and the name of the structure (here: `yPDef_SOut`) must be used in the `xOutEnv` C function in this case.

The code generation of structure types and signal IDs is (except the C comment about `IN` or `OUT`) independent from the direction the signal goes.

Process IDs

At last the process ID declarations are generated as `#define` values in C, like:

```
#define XPTID_<auto-prefix>_MyProcess 0
```

where the first process in the system is the value of 0 assigned, the second process gets the value 1, and so on. Due to the implementation of SDL'92 object orientation in the Cmicro SDL to C Compiler, there is also an automatic prefix generated. Using this prefix in the user's environment functions, it is possible to distinguish between several processes with the same name. Please refer to "Generation of Identifiers" on page 3354 for more information.

Generated Make File

The Cmicro SDL to C Compiler generates a file that contains production rules for the C program. This file can be used together with "make" facility only. The file is called `<systemname>.m`.

Additionally there is an ASCII file called `<systemname>_gen.m` which gives a list of all the generated files. This files is used by the Targeting Expert to generate a makefile. Please see "Generated Makefile" on page 2922 in chapter 60, *The Targeting Expert*.

Generated Symbol File

The generated symbol file is used to store symbolic information about the SDL system. The file has meaning for the host part of the SDL Target Tester only and is called `<systemname>.sym`. It is used for SDL Target Tester purposes only and is described within "The Host Symbol Table" on page 3637 in chapter 68, *The SDL Target Tester*.

Generated Kernel Group File

The generated kernel group file contains information about process names. This file is especially used in integrations, when OO is used and processes are instantiated. Using the information from this file, it is pos-

sible to distinguish between several process instantiations with the same name.

Implementation

In this section the implementation details are discussed. These details are meaningful for understanding how a generated Cmicro application does work.

Time

For host simulation, with the predefined integration settings, a time unit represents one second. In target applications, time is to be implemented by the user (see subsection "Defining the SDL System Time Functions in mk_stim.c" on page 3435 in chapter 67, *The Cmicro Library*).

Real Time

If real time is used, then there will be a connection between the clock in the executing program and the wall clock. For applications the user must provide the connection with the wall clock, normally the hardware timer.

> **Note:**
>
> The C standard function `time` used as the real time clock returns the time in seconds. The implementation of the clock can be changed by re-implementing the function `xmk_NOW` in `mk_stim.c`.

Scheduling

The Cmicro Kernel does not use a process ready queue. It processes the signals in the order of their appearance. To do this, there is a signal queue which stores the signals sent to any process (either internally or externally). There are different ways to influence the scheduling when using the Cmicro SDL to C Compiler:

- assigning priorities to processes
- assigning priorities to signals
- any combination of process and signal priorities

Assigning Priorities to Processes – Preemptive Scheduling

It is possible to assign priorities to process types (using SDL'88 terminology). The processes' priorities are assigned when designing the SDL system. They are assigned using the #PRIO directive.

There are some things to be kept in mind when using process priorities:

- Priorities have to begin with zero.
- Priorities have to be consecutive.
- All instances of a type have the same priority (SDL'88 terminology).
- Priority decreases with increasing numbers (zero is the highest priority level).
- The default priority is to be in the range of zero to the lowest priority number.

The Cmicro Kernel handles process priorities by collecting all signals sent to processes of the same priority in a separate queue. Thus, there is a queue for each priority level.

While the SDL system is running the kernel checks for signals in the queues with decreasing priority. This check takes place whenever an SDL output appears or a process performs an SDL nextstate operation. Because of the kernel checking for signals whenever an output takes place, it is possible to have preemptive scheduling.

Assume, there are two process types lowprio and highprio. Let process type lowprio have the priority one and process type highprio have the priority zero.

If an instance of process type lowprio performs an output to process type highprio, there appears a signal in a queue of a higher priority level (zero is the highest priority level available, process lowprio has priority one) which leads to the kernel immediately working on the signal sent to the process highprio. The transition of process lowprio will not end until process highprio has finished its transition invoked by the signal.

This way of scheduling is implemented using recursion.

Note:

Process priorities are available only when using a compiler which can handle recursion.

There is basically no restriction on the number of priority levels, but the target and compiler used will of course limit the depth of recursion.

As a general recommendation process priorities should not be assigned one per process type, but the process types should be grouped according to their purposes and these groups should then be assigned a priority level.

Assigning Priorities to Signals

The signals in the queue(s) are normally ordered according to their appearance (FIFO-strategy). By assigning priorities to signals this ordering is user definable. The directive #PRIO is used to assign a priority to signals.

Priority increases with decreasing numbers, but there is no restriction to use consecutive numbering.

Whenever a signal is sent, it is inserted into the signal queue(s) according to its priority.

Assume, there is a process performing two signal outputs, first_sig and second_sig. Using the standard FIFO-strategy signal first_sig would be worked on before signal second_sig. But with signal priorities and signal first_sig assigned priority fifty and signal second_sig assigned priority twenty, signal second_sig would be in front of signal first_sig in the queue and thus would be worked on before signal first_sig.

For more details please refer to "Assigning Priorities – Directive #PRIO" on page 3323.

Combinations of Signal/Process Priority

Every combination of signal and process priorities may be used. In this way it is possible to adapt the scheduling to the users' needs.

Note:

Without process priorities a transition once started will have to be finished before the next transition can be dealt with. This is valid regardless of the time it will need to finish a transition.

Synonyms

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 must be included as macro definitions into the generated code, for instance by including another header file.

Using a Macro Definition

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

1. Write the actual macro definitions on a file.

   Example 543: Macro Definition

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

   The synonym names are the SDL names (without any prefixes).

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

   Example 544: #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.

Procedure Calls and Operator Calls

In SDL-92, value returning procedures and operator calls are introduced. This means, that an SDL procedure can be called within an expression. As the Cmicro SDL to C Compiler cannot handle procedures with states, it is not necessary to map such calls to a different scheme.

Example 545: Procedure Call

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

is translated to something like:

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

Note:

The value returning procedure calls are transformed to C functions which return values.

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

Generation of PAD function

The code generation for the PAD function is different compared with Cadvanced, in the way that code that is common in process types is copied into the PAD function for instantiated processes. This is implemented in contrast to Cadvanced, where for each process type definition there is a C function generated once, that is called by the instantiated PAD function, for common code. This makes a difference when system partitioning and/or file separation is used.

Any

‘Any’ should not be used in applications using the Cmicro SDL to C Compiler, as it leads to an error message.

Calculation of Receiver in Outputs

The Cmicro SDL to C Compiler is a code generator using the semantics of SDL-92 with some restrictions. The behavior for output is according to the rules described in the following:

- For an output without TO and without VIA in SDL, the Cmicro SDL to C Compiler calculates the receiver of the signal during code generation. If there is more than one possible receiving process type, then an error message will be printed out.
- For an output without TO and without VIA in SDL, it is also possible to have one process type, but more than one receiving instance of the signal. The response is that any of the living possible receivers may be selected during execution time. If no receiver is found, the C function `ErrorHandler` will be called.
- For an output with the VIA clause, the behavior of the Cmicro SDL to C Compiler is in principle the same as for an output without TO It computes the possible receivers in an output with the VIA clause and if there are several possible receivers, an error message is produced. The only difference between output with VIA and output without TO is that VIA can restrict the amount of possible processes.
- If output with TO is used in the above cases, no ambiguity can occur. The addressing of the process is then performed by a run-time variable.
- The possibility of specifying the name of a process when using OUTPUT TO is implemented. This is an SDL-92 feature. The Cmicro SDL to C Compiler behaves in the same way as when using implicit addressing (output without to).
- The broadcast feature of SDL-92 (VIA ALL) is not implemented, because it is not a real broadcast and not very useful for Cmicro Applications.

Abstract Data Types

In this section the specialities and exceptions about abstract data types for Cmicro are discussed only. A complete documentation about the abstract data types is given in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*.

General C Definitions

All the macros and external definitions for functions can be found in the file `sctpred.h` except for the PId sort which is handled in the file `ml_typ.h`.

The C functions for the handling of predefined sorts are defined in the file `sctpred.c`.

On UNIX these files can be found in `$sdtdir/cmicro/kernel`.

In Windows these files can be found in the Telelogic Tau installation under `%SDTDIR%\cmicro\kernel`.

Exceptions for SDL Predefined Types

A general exception existing for all the predefined types is that the user must configure which predefined types are to be compiled into the target C program. This is necessary to hold the target C program as small as possible. The configuration is to be performed with the help of the Targeting Expert, please view "Configure and Scale the Target Library" on page 2872 in chapter 60, *The Targeting Expert*.

Caution!

Problems will occur during compilation when the configuration is not according to what the SDL system needs. The user should refer to the explanations about manual scaling in chapter 67, *The Cmicro Library*.

External Synonyms

External synonyms are to be defined by the user in the following way.

For a synonym like

```
synonym xternal integer = EXTERNAL;
```

Cmicro expects to see `xternal` as a `#define` value that is to be defined by the user. This can be done for example in the following way:

```
synonym xternal integer = EXTERNAL;
/*#CODE
#TYPE
#ifdef XSCT_CMICRO
#define xternal 7
#endif
*/
```

This also means, that if `xternal` is not defined from the user, it will lead to compilation errors.

Charstring

Charstrings can be used either in the usual way as they are when using Cadvanced, or they can be used in a restricted way. The decision is up to the user and is a question of configuration. The user should be aware that some of the predefined sorts from ASN.1 are based on the implementation of SDL charstrings. This is discussed in subsection "Support of SDL Constructs" on page 3417 in chapter 67, *The Cmicro Library*.

Time/Duration

The predefined data types Time and Duration are implemented in a more or less restrictive way. It is possible to specify a real value for Time and Duration on SDL level, like 23.45. The Cmicro Library uses only the integer part in front of the dot, 23 in this example. The mapping of SDL time units to time units in a target application is – in any case – up to the user.

UnionC

The #UNIONC directive is not recommended when using the Cmicro SDL to C Compiler because there is no support for checking the validity of the component selection. Both the #UNION directive and the CHOICE concept are a better alternative.

Predefined Generators Array, String, Powerset, Bag, Ref

These generators are implemented in Cmicro, but the user should be aware that the use of any of them requires that dynamic memory allocation is used in the target system. Generally, Cmicro tries to prevent the use of dynamic memory allocation whenever possible. The reasons for this are explained in chapter 67, *The Cmicro Library*.

ctypes.sdl

This package can be used together with Cmicro with the following restriction.

There are two operators that are excluded when Cmicro C code is compiled. The operators are "`CStar2CString`" and "`CharStar`".

The reason for this is that with Cmicro it is possible to define an array of char in C instead of the predefined solution of Cadvanced (to use dynamic memory allocation). This is discussed in subsection "Support of SDL Constructs" on page 3417 in chapter 67, *The Cmicro Library*.

byte.pr

This ADT can be used together with Cmicro in the same way as described for Cadvanced.

file.pr

This ADT is not useful for typical Cmicro applications (embedded systems usually do not provide a hard disk in Cmicro applications) and for that reason never has been tested. The ADT may however work with Cmicro.

idnode.pr

This ADT **cannot** be used together with Cmicro because it refers to Cadvanced code.

list1/list2.pr

This ADT **cannot** be used together with Cmicro because it refers to Cadvanced code.

long_int.pr

This ADT can be used together with Cmicro.

pidlist.pr

This ADT **cannot** be used together with Cmicro, generally.

Instead of `pidlist.pr`, the user may include the `cm_pidlist.pr` file. The use of this ADT is however restricted, because Cmicro implements a different scheduling algorithm. This means, that systems that

are successfully simulated first, may contain problems when a Cmicro target application is build and executed.

It is therefore recommended **not** to use neither `pidlist.pr` nor `cm_pidlist.pr`, in order to achieve the best possible SDL conformity.

random.pr

This ADT **cannot** be used together with Cmicro because it refers to Cadvanced code.

unsigned.pr

This ADT can be used together with Cmicro.

unsigned_long.pr

This ADT can be used together with Cmicro.

Default Values

Default values are in principle generated in the same way as with the Cadvanced SDL to C Compiler. It is however possible to configure the default value setting, which is explained in chapter 67, *The Cmicro Library*. The right configuration is essential to prevent illegal behavior.

Exceptions for Implementations of Operators

Read and Write Functions

The Cmicro SDL to C Compiler does not provide read and write functions. The reason is, that the Cmicro SDL to C Compiler mainly is used to build target applications, and not simulations. This is also a consequence of optimizing the target program. If the user uses the Q (question) operator, the Cmicro SDL to C Compiler ignores this.

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 `ErrorHandler`, with the following prototype

```
extern void ErrorHandler( xmk_OPT_INT errnum )
```

can be used for this purpose. `xmk_OPT_INT` is defined in `ml_typ.h`, normally as an ordinary C `int`. `errnum` may be one of the free values of error numbers. Please inspect `ml_err.h` in order to get a list of reserved values.

Example 546: Error Handler in Operator

```
  if ( strlen(C) <= 1 ) {
#ifdef XMK_USE_ERR_CHECK
    ErrorHandler (ERR_N_InvalidStringLength);
#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 C function `ErrorHandler` is called and a default value is returned (NULL). By including the call to `ErrorHandler` between `#ifdef XMK_USE_ERR_CHECK - #endif` the function is only called to report the error, if error checks are turned on. The one parameter to the C function `ErrorHandler` should identify the error. The number must be given by the user.

Another possibility to route error messages to the host system is to use the C function `xmk_PrintString` of the SDL Target Tester, defined as:

```
extern void xmk_PrintString( char * )
```

Example 547: Error Handler in Operator

```
  if ( strlen(C) <= 1 ) {
#ifdef XMK_ADD_MICRO_TESTER
    xmk_PrintString ("ERR:Invalid Stringlength");
#endif
    return SDL_NUL;
  } else
    return C[1];
```

Access to Predefined Sorts based on Charstring

As already mentioned in earlier subsections, the user should be aware that some of the ASN.1 predefined sorts are based on the implementation of SDL charstrings. The user should also refer to subsection "Sup-

port of SDL Constructs" on page 3417 in chapter 67, *The Cmicro Library*.

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 run-time library that might be helpful when handling Charstrings.

It is usually necessary to allocate dynamic memory when an operator returning a charstring value is implemented. There are two help functions that should be used in connection with allocation and de-allocation of dynamic memory. These are documented in "Dynamic Memory Allocation" on page 3450 in chapter 67, *The Cmicro Library*.

Caution!

Do not use Charstring in SDL if you want to get a correct trace output with the SDL Target Tester, or if you want to use the Cmicro Recorder. In the last case, the use of charstring may lead to a fatal error when an SDL session is replayed.

Exceptions for Directives

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 command in chapter 2, *The Organizer*. Here file names for the generated files can also be specified. In the *Make* dialog in the Organizer (see "Generated Files" on page 3302) it is possible to select full separate generation, user-defined separate generation, or no separate generation.
- For an SDL/PR file that is generated by 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 file generation, user-defined separate file generation, or no separate file 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 Cmicro SDL to C Compiler can generate a separate file for:

- System (always separate)
- Block
- Process
- Procedure

Note:

Instantiations cannot be separated. 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 548: #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 Cmicro SDL to C Compiler 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 a case the file name becomes the name of the unit with the appropriate extension (`.c .h`) depending on contents. The file name is stripped of 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 "Generated Files" on page 3302), can be used in a simple manner

to select certain default separation schemes. This setting will be interpreted in the following way:

- *No* separation.
  The whole system will be generated into one 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, and process, and process 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 been selected, the Cmicro SDL to C Compiler will use the file name specified in the *Edit Separation* 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 options.

Example 549: #SEPARATE Directive ––––––––––––––––––––––––

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

––

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 re-compile 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, using the Organizer *Edit Separation* command or 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.

Assigning Priorities – Directive #PRIO

#PRIO for Processes

Priorities can be assigned to processes using the directive #PRIO. The process priorities will affect the scheduling of processes, see "Scheduling" on page 3377. A priority is a positive integer, where low value means high priority. #PRIO directives should be placed directly after the process heading in the definition of the current process.

Example 550: #PRIO Directive

```
Process P1; /*#PRIO 0 */
Process P2(1,1); /*#PRIO 1 */

Process P3 : P3Type; /*#PRIO 0 */
Process P4(1,1) : P4Type; /*#PRIO 1 */
```

Processes that do not contain any priority directive will have a user defined default priority with the name `xDefaultPrioProcess`.

There are some things to be kept in mind when using process priorities:

- Priorities have to begin with zero.
- Priorities have to be consecutive (0,1,2,3,4,5).
- All instances of a type have the same priority.
- Priority decreases with increasing numbers (zero is the highest priority level).
- The default priority is to be in the range of zero to the highest priority number, that is 0 or 1 in the example above.

#PRIO for Signals

Priorities can be assigned to signals using the directive #PRIO. The signal priorities will also effect the scheduling of processes, see "Scheduling" on page 3377.

Signal priorities can be specified, either:

- in the declaration of the signal
- in SDL output

It is impossible to specify #PRIO in a SDL input. Cmicro will ignore any occurrence of #PRIO in SDL input.

Signal priorities do affect the SDL output and the SDL create actions only.

The following rules are to be considered here:

- Signal priorities have to be in the range from 0 to 255.
- Signal priorities do not have to be consecutive, as process priorities have to be.
- If not specified otherwise (in SDL output) all instances of a signal have the same priority.
- Signal priority decreases with increasing numbers (zero is the highest priority level).
- The signal default priority is to be specified by the user (`xDefaultPrioSignal`), and must in the range of 0 to 255. As a recommendation, this value should be set to 100, so that both higher, as well as lower priorities can be declared with #PRIO.
- Signal priorities come after process priorities.
- If no #PRIO is specified for a signal, neither in its declaration, nor in any output, Cmicro uses `xDefaultPrioSignal` for each occurrence of that signal in an output.
- If #PRIO is specified only in the declaration of a signal, Cmicro uses this specified priority in each occurrence of that signal in an output.
- If #PRIO is specified in a specific output of a signal, but not in its declaration, then the specified #PRIO value is taken from the out-

put. If the signal is output without #PRIO, in that case `xDefaultPrioSignal` will be used.

- For dynamic process creation, an internal create signal is used. This signal carries the priority defined by `XMK_CREATE_PRIO`. As a general recommendation, this priority should be higher than any other signal priority.

The following example will give more explanations (note, that the values PA, PB are of sort pid):

Example 551: #PRIO Directive

```
Signal
S1, /*#PRIO 11 */
S2, /*#PRIO 22 */
S3,
S4; /*#PRIO 44 */

....
output S1 to PA
output S1 to PB; /*#PRIO 55*/
....
output S3 to PC;
output S3 to PD; /*#PRIO 66*/
```

Assuming the following C definition:

```
#define xDefaultPrioSignal 100
```

the following priorities will then be generated:

```
output S1 to PA--->use of prio 11
output S1 to PB--->use of prio 55
....
output S3 to PC--->use of prio 100
output S3 to PD--->use of prio 66
```

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

There is no difference for the #EXTSIG, #ALT and #TRANSFER directives for Cmicro compared with Cadvanced, except that the use of it will sometime lead to a better performance. This is because if #EXTSIG for example is used in the case of an output to the environment, the user can prevent the Cmicro Kernel to be called (and the `xOutEnv` function to be executed).

Output of Code Generation

This section gives an overview of the code generated by the Cmicro SDL to C Compiler. This is useful, to make it possible to interpret the generated code. To know how the code is generated makes it quite easy to understand the program which is necessary and useful when testing and debugging erroneous executable programs.

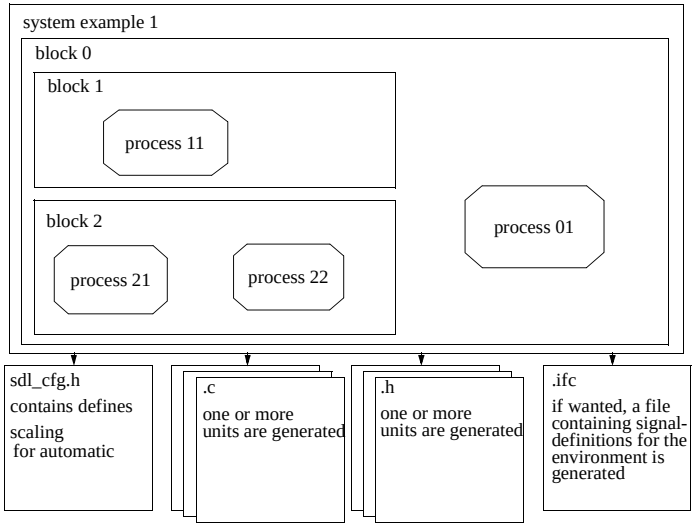

Figure 577: Structure of the generated C code

Not all the intricate details of the generated code are described here. The depth of description is sufficient to give the reader a reasonable understanding of the code generation algorithms. Explanations will illustrate what the code looks like, but not why.

The generated code contains several places where prefixes are generated, which consists of a prefix and unique numbering. The following prefix is generated for all objects: "`z<nnn>_`", where nnn is an incremental number.

Allowance for conditional compilation occurs in several places throughout the generated code. The generated C code is conditionally compiled, for example, for dynamic process creation (create symbol). A differentiation is made between conditional compilations generated by

the Cmicro SDL to C Compiler (called automatic scaling, prefix XMK_USED_) and conditional compilations which are dependent on header files, which are to be modified by the user (called manual scaling, prefix XMK_USE_).

Note:

Generally speaking, the ordering of the following subsections corresponds to the ordering in which the code is generated.

Each compilation unit is compiled either in one a.c file or into two files, a.c and a.h.

Only the differences are shown, when comparing the output of SDL to C Compiler with the Cmicro SDL to C Compiler. The overall differences of the both code generators are described in the section "Differences between Cmicro and Cadvanced" on page 3364 in chapter 67, *The Cmicro Library*.

Header of Generated C File

Code generation on the .c file for the current unit is started by generating the following header:

Example 552: The Head of a Generated C File

```
/* Program generated by the SDL suite.Cmicro,
version x.y */
#define XSCT_CMICRO

#define C_MICRO_x_y
#define XMK_C_TIME <GenerationTime>
#include "ml_typ.h"
```

The XSCT_CMICRO macro can be used by the user to distinguish between the different Code generators, for example within ADT bodies.

The C_MICRO_x_y macro can be used by the user to distinguish between different versions of the Cmicro SDL to C Compiler. This is usually not but might become necessary if the output of the Cmicro SDL to C Compiler is different.

The XMK_C_TIME macro is used internally when compiling and linking and executing with the SDL Target Tester takes place. With this macro, a rough consistency check for the generated files is done. The

<GenerationTime> of the different files that are generated is compared in the Library and by the SDL Target Tester. If there is an inconsistency, compilation errors will occur.

The #include "ml_typ.h" is used to include all necessary declarations that the generated C code may use, including automatic scaling from sdl_cfg.h and predefined sorts.

SECTION Types and Forward References

As a difference to SDL to C compiler, this section contains the definitions for the process IDs and the forward declarations used in the generated C code.

Process IDs are generated as #define values in C, like:

```
#define XPTID_<UniquePrefix>_MyProcess 0
```

where the first process in the system is the value of 0 assigned, the second process gets the value 1, and so on. Please refer to "Generation of Identifiers" on page 3354 for more information.

The following forward references are generated:

```
extern XCONST XPDTBL yPDTBL_<UniquePrefix>_MyProcess;
```

Following this, the usual declarations are generated as described in *chapter 57, The Cadvanced/Cbasic SDL to C Compiler*.

No synonym variables are generated when using Cmicro.

Symbol Tables

Symbol tables are only generated for the SDL Target Tester, and not into the generated C code. The symbol tables generated for the SDL Target Tester are described within chapter 68, *The SDL Target Tester*.

Tables for Processes

Tables are used to represent the behavior of SDL objects, like processes and timers. It is not absolutely necessary to understand how these tables are generated and how the Cmicro Kernel works with them. The following subsections are only for those readers interested in the nature of the table structure.

Root Process Table

The root process table contains, for each of the defined SDL process types, a reference (i.e. a pointer) to the Process Description Table. The Cmicro Kernel is the main user of the root process table. Via this table, it can access all SDL process types and all SDL process instance data. The location of the generated root process table is directly before the yPAD-functions in the generated C file. The type definitions used in this table are located in the `ml_typ.h` module.

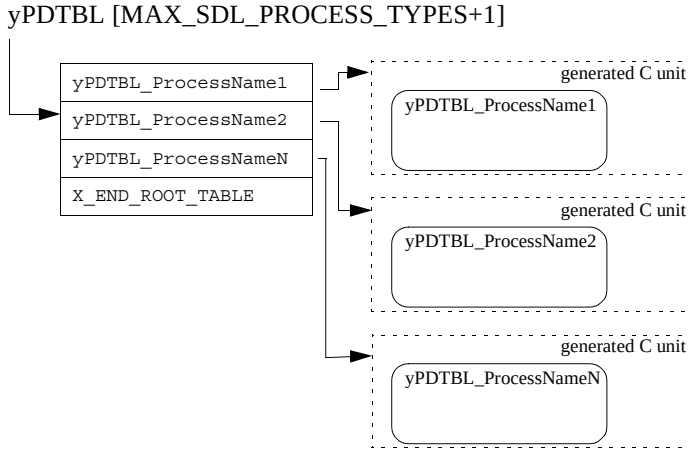

Figure 578: Root process table

Example 553: Code of Root Process Table ––––––––––––––––––––

C-Type definition (`ml_typ.h`):

```
extern xPDTBL yPDTBL [];/* for the Cmicro Kernel */
#define X_END_ROOT_TABLE/* Table-End Marker of yPDTBL*/
```

C constants (`sdl_cfg.h`):

```
#define MAX_SDL_PROCESS_TYPES <N>

   /* <Process-type-id´s>  Process Types are numbered */
   /* from 0 to N-1(see chapter "Generating PID") */
   #define XPTID_Process1Name 0
   #define XPTID_Process2Name 1
   #define XPTID_ProcessnName N-1
```

C code generation for the whole system:

```
XPDTBL yPDTBL [MAX_SDL_PROCESS_TYPES+1] =
{
  yPDTBL_ Process1Name,
  yPDTBL_ Process2Name,
  ......
  yPDTBL_ ProcessnName,
  X_END_ROOT_TABLE
}
```

Symbol Trace Table

In order to reduce the use of dynamic memory allocation, there is a table generated in the code which is used by the SDL Target Tester to store and retrieve test options, like switches, which define the trace.

The table is conditionally compiled and only included if the SDL Target Tester is contained in the target- executable.

The symbol trace table looks like:

Example 554: Code for Symbol Trace Table

```
/************************************************************
**  Symbol trace table
************************************************************/
#ifdef XMK_ADD_TEST_OPTIONS
XSYMTRACETBL *xSYMTRACETBL[MAX_SDL_PROCESS_TYPES+1] =
{
  (XSYMTRACETBL_ENTRY *)  NULL, /* for first Processtype */
  (XSYMTRACETBL_ENTRY *)  NULL, /* for second Processtype */
  ........
  ........
  (XSYMTRACETBL_ENTRY *)  NULL, /* for last Processtype */
  X_END_SYMTRACE_TABLE /* table end marker */
};
#endif
```

More information can be obtained by reading chapter 68, *The SDL Target Tester*.

Optimized Decision Trace information

An option to reduce the trace information for SDL decisions by showing only the first ten characters of the decision expression during trace. Setting the environment variable CMICRO_SHORT_DECISION_TRACE to any value prior to the Cmicro code generation is started, will have the effect on the generated C code that all xTraceDecision<parameter> statements will contain a parameter that is the first ten characters of the

decision expression instead of the complete expression. This will reduce the trace information for systems that contain a lot of decisions.

Instance-Data-Struct

The struct is generated in the header-section of the generated C file.

Example 555: Code Generation of type definition for each SDL process

```
typedef struct {
  PROCESS_VARS
  TypeName1  FPAR_var1;
  TypeName2  FPAR_var1;
  TypeName3  DCL_var1;
  TypeName4  DCL_var2;
  TypeName4  yExp_DCL_var2;
  TypeName5  FPAR_var1;
} yVDef_ProcessName;
```

Instances of a given type are represented as a C array. The code generation of variables for each SDL process looks like:

Example 556

```
#define X_MAX_INST_ProcessName upperlimitofprocessinstances1

static yVDef_ ProcessName

yINSTD_ProcessName[X_MAX_INST_ProcessName];
```

A reference to this array is generated in the Process Description Table which is discussed in the subsection "Process Description Table" on page 3335.

Process State Table

This table is generated for each process in the header-section of the generated C file. It contains information about the state of each process instance. The table contains ordinary SDL state values as well as the values `XSTARTUP` and `XDORMANT`. `XSTARTUP` is generated for each instance which is to be statically created (in (x, N) declarations, where x is > 0), `XDORMANT` is the value which is used to tag a process instance as sleeping. In the case of creation this instance can be reused.

Example 557: Code for Process State Table ––––––––––––––––––––

C typedef for the process state table (located in `ml_typ.h`):

```
typedef u_char xSTATE; /* see defines below */
#define XSTARTUP 0xff  /* valid only if xSTATE is */
                       /* u_char else 0xffff */
#define XDORMANT 0xfe  /* valid only if xSTATE is */
                       /* u_char, else 0xfffe */
```

C code generation for each process:

```
static xSTATE yPSTATETBL_znn_ProcessName
                                  [X_MAX_INST_znn_ProcessName] =
{
  <creation-tag>  /* Instance 0 */
  <creation-tag>  /* Instance 1 */

  <creation-tag>  /* Instance M-1 */
};
```

where `<creation-tag>` is either `XSTARTUP` or `XDORMANT`.

––

Example 558: ––

Code for a process type with 4 instances, 2 of which are to be created at SDL system start:

```
static xSTATE yPSTATETBL_znn_ProcessName [4] =
{
  XSTARTUP, /* Create at SDL-system-start */
  XSTARTUP, /* Create at SDL-system-start */
  XDORMANT, /* Create later               */
  XDORMANT  /* Create later               */
};
```

––

A reference to this table is created in the Process Description Table, which is discussed in the subsection “Process Description Table” on page 3335.

Transition Table

This is generated in the header-section of the generated C file. It contains all transitions of a process, including asterisk states, asterisk inputs and asterisk save.

The C `typedef` for the transition table (located in `ml_typ.h`) is as follows:

Example 559: Code for Transition Table ––––––––––––––––––––––

```
typedef struct {
  xINPUT    SignalID; /* Input, Asterisk-Input. */
                      /* Input is Timer */

                      /* and/or ordinary Signal */
  xSYMBOLNR SymbolNr; /* Symbolnumber to be used */
                      /* in yPAD-function */
} xTR_TABLE_ENTRY;
```

C code generation:

```
static XCONST xTR_TABLE_ENTRY   yTRTBL_znn_ProcessName
                                [XMAX_TRANS_znn_ProcessName]=
{
  /* state_0-table */
  input_1, SymbolNr,
  input_2, SymbolNr,
  XASTERISK,XSAVEID /* asterisk save */

  input_N, SymbolNr,

  /* state_1-table */
  ..............
  ..............
  /* state_j-table */
  input_1, SymbolNr,
  input_2, SymbolNr,

  input_N, trans_jN,
  XASTERISK,XSAVEID /* asterisk save */
};
```

The `SymbolNr` shown above is used to select the right transition in the switch generated in the yPAD function.

Where the C define

`XASTERISK` is an ID defining all possible SDL Inputs (asterisk Inputs),

`XSAVEID` is a simple ID defined in `ml_typ.h` which can be compared by the SDL Kernel to detect signal-save.

And where:

```
#define XASTERISK   -1
#define XSAVEID     xSave
```

––

A reference to this table is created in the Process Description Table.

State Index Table

This is generated in the header section of the generated C file.

Example 560: Code for State Index Table

C typedef (`ml_typ.h`):

```
typedef   u_char xSTATE_INDEX;
```

C code generation (header of generated C file):

```
static xCONST xSITBL   xSTATE_INDEX_znn_ProcessName
                         [<count_transitions_of_ProcessName] =
{
  0, /* i.e.a process with 3 states, but no asterisk states */
     /* state_0 has 2 transitions */
  2, /* state_1 has 5 transitions */
  7, /* state_2 has 3 transitions */
  10 /* table-end-index XI_TABLE_END */
};
```

The first value in the above table indicates the beginning of the first state in the Transition Table. If asterisk state definitions are not found in the process, this value is 0.

A reference to this table is created in the Process Description Table.

PID Table

These tables are used to store the values parent and offspring for each process. The reason an extra table is used to store this information is to simplify initialization. The Cmicro Kernel updates the values in the table according to the SDL rules.

Example 561: Code for PID Table

C-type definition (`ml_typ.h`):

```
#ifdef XMK_USE_PID_ADDRESSING
  typedef struct
  {
    #ifdef XMK_USE_SDL_PARENT
      xPID            Parent;
    #endif

    #ifdef XMK_USE_SDL_OFFSPRING
      xPID            Offspring;
    #endif

  } xPIDTable;
#endif
```

C code generation for each process:

```
/*-----------Process-PID-Values-------------*/
#ifdef XMK_USE_PID_ADDRESSING
  static xPIDTable yPID_TBL_z00_P1[X_MAX_INST_z00_P1];
#endif
```

A reference to this table is created in the Process Description Table, which is discussed in the subsection "Process Description Table" on page 3335.

Process Description Table

For each SDL process, an automatically initialized C structure is generated called process description table. This table is used in the Root Process Table to enable the Cmicro Kernel to access process type information as well as process instance data.

Inspect the following diagram to see which information is contained in the process description table:

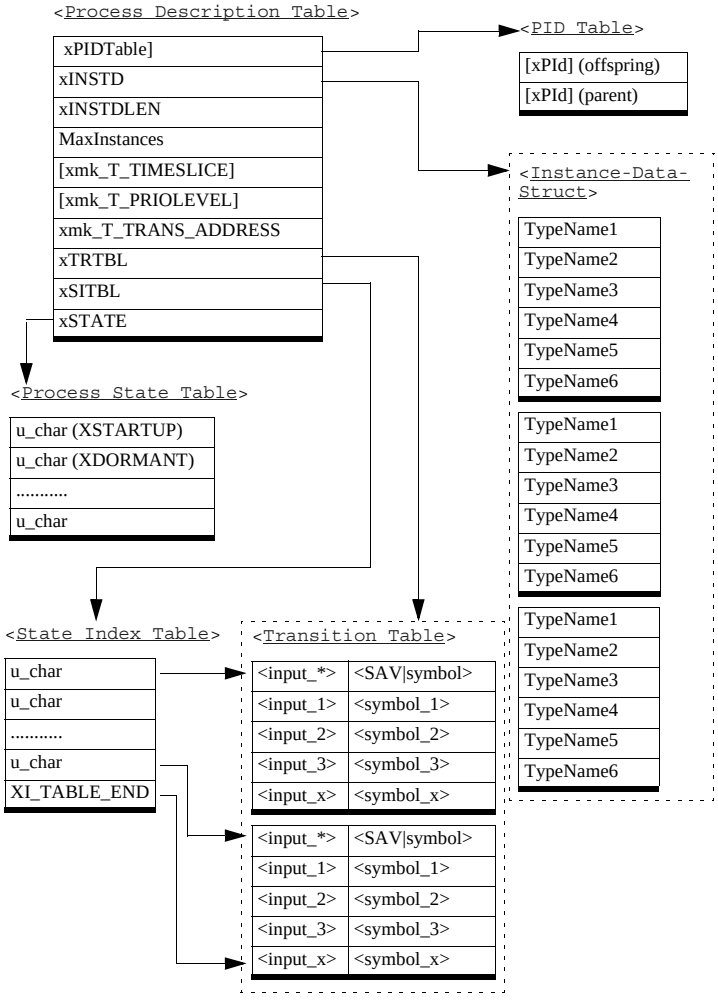

Figure 579: Process description table

Allocated to each SDL process type is one table `yPDTBL_ProcessName`.

The type definitions of this table are located in the `ml_typ.h` module.

Example 562: Code for Process Description Table

C typedef for the process description table (`ml_typ.h`):

```
typedef struct {
  #ifdef XMK_USE_PID_ADDRESSING
    xPIDTable  *pPIDTable; /* Table with */
                           /* Parent/OffspringValues */
  #endif

  xINSTD      *pInstanceData ; /* Pointer to Instancedata*/
                               /* Vector                 */
  xINSTDLEN    DataLength ;  /* Length of Instancedata  */
                               /* for 1 Instance         */
                               /* (used by SDL-BS)       */
  unsigned char MaxInstances ; /* Max.Number of Instances*/

  #ifdef XMK_USE_TIMESLICE
    /* Time-Slices can be individually specified by the user*/
    /* The value stored in TimeSlice is measured in ticks   */
    /* The Cmicro Kernel has to be scaled to handle     */
    /* timeslicing */
    xmk_T_TIMESLICE TimeSlice;
  #endif

  #ifdef XMK_USE_PREEMPTIVE
    /* Process-Priority can be specified with #PRIO on the */
    /* SDL-Level. It is available only, if the Cmicro    */
    /* Kernel is scaled to handle preemption. */
    xmk_T_PRIOLEVEL PrioLevel;/*Priority of this processtype*/
  #endif

  xmk_T_TRANS_ADDRESS yPAD_Function ; /* Address of the */
                                      /* yPADFunction */
  xTRTBL TransitionTable ;  /* Pointer to transition table */
  xSITBL *StateIndexTable ; /* Pointer to state index table */
  xSTATE *ProcessStateTable;/* Pointer to process state table
*/
} XPDTBL;
```

C code generation for each process:

```
#define X_MAX_INST_ProcessName 1

xPDTBL yPDTBL_ProcessName =
{
  yPID_TBL_znn_<process:N>,
  (xINSTD*) yINSTD_znn_ProcessName,
  X_MAX_INST_znn_ProcessName,
  (xmk_T_TRANS_ADDRESS) yPAD_znn_ProcessName,
  yTRTBL_znn_ProcessName;
  xSTATE_INDEX_znn_ProcessName,
  yPSTATETBL_znn_ProcessName;
};
```

For each generated process description table, a new entry in the Root Process Table is generated.

Actions by Processes and Procedures

GR References

No code is generated to evaluate the graphical references during runtime of the SDL system. A large amount of memory is required to store and handle such information which normally proves too large for any real target system.

Alternatively, C comments are generated which make it possible to verify and debug the generated code as illustrated in the following example. The PR <position> indicates in which line number of the SDL/PR file the symbol can be found.

```
For processes :
/*************************************************************
**  PROCESS <process-name>
**  <<SYSTEM <system-name>/BLOCK <block-name>>
**  #SDTREF(<reference>)
*************************************************************/

For signals :
/*************************************************************
**  SIGNAL S1
**  <<SYSTEM <system-name>/BLOCK <block-name>>
**  #SDTREF(<reference>)
*************************************************************/

For yPAD-function
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
** Function for process <process-name>
** #SDTREF(<reference>)
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

For output :
/*-----
** OUTPUT <signal-name>
** #SDTREF(<reference>)
------*/

For nextstate :
/*-----
** NEXTSTATE <state-name>
** #SDTREF(<reference>)
------*/
```

Structure of Process and Procedure Functions

The basic structure of the generated C code for process and procedure definitions remains the same as for the SDL to C compiler although some modifications are evident.

The code generation for the PAD function is different compared with Cadvanced, in the way that code that is common in process types is copied into the PAD function for instantiated processes.

Procedures follow the same code generation as processes, with some small exceptions in macro naming conventions for variable declarations.

Each SDL process is represented in C by a C function called `yPAD_ProcessName`.

Example 563: `yPAD_ProcessName` ----------

```
/* Function for process ProcessName */
#ifndef XNOPROTO
extern YPAD_RESULT_TYPE yPAD_ProcessName ( YPAD_ANSI_PARAM )
#else
extern YPAD_RESULT_TYPE yPAD_ProcessName ( YPAD_KR_PARAM )
  YPAD_KR_DEF
#endif
{
  local variable section
  State-input-selection
  {
    start-transition including nextstate
    transition-1 including nextstate
    transition-2 including nextstate
    .......
    transition-n including nextstate
  }
  pad-end-section
}

/* Function for procedure ProcedureName */
#ifndef XNOPROTO
extern YPRD_RESULT_TYPE yPAD_ProcedureName ( YPRD_ANSI_PARAM )
#else
extern YPRD_RESULT_TYPE yPAD_ProcedureName ( YPRD_KR_PARAM )
  YPRD_KR_DEF
#endif
{
  local variable section
  section representing procedure body
}
```

Local Variables Section

The following defines are generated in the local variables section for processes.

Example 564 ----------

```
YPAD_YSVARP             /* used for signal variable pointers
*/
YPAD_YVARP(yVDef_z00_P1)/* used for process variables */
YPAD_TEMP_VARS          /* used for temporary variables */
YPRSNAME_VAR("P1")      /* can be used for printf */
BEGIN_PAD               /* used for some preparations */
                        /* to handle signals, or Integration*/
                        /* of any Realtime operating system */
```

After expansion by the C preprocessor:

```
yVDef_z00_ProcessName *yVarP
                          =(yVDef_z00_ProcessName *)pRunData;
unsigned char *yOutputSignal;
unsigned char *ySVarP;

(void) printf(("PROCESS:%s\n", "ProcessName"));

if((P_MESSAGE != ((void *) 0))
    && (P_MESSAGE->mess_length > 4))
{
  ySVarP = (unsigned char *) P_MESSAGE->mess_ud.pt_ud;
}
else
{
  ySVarP = (unsigned char *) P_MESSAGE->mess_ud.ud;
}
```

The following defines are generated in the local variables section for procedures:

```
YPRD_YVARP(yVDef_znnn_ProcedureName)
/* used for procedure variables */

YPRD_TEMP_VARS
/* used for temporary variables */

YPRDNAME_VAR("ProcedureName")
/* can be used for printf */
```

State – Input Selection

The selection of the appropriate SDL transition which is to be executed in the current state with the current signal in the input port goes in principle over the transition table, described in previous chapters. With this table, the Cmicro Kernel can evaluate a symbol number, which is local to a process, a unique numbering of the different possible transitions. This numbering algorithm begins at 0 (which corresponds to the start symbol) and continues until all symbols for this particular process type have been numbered.

The appropriate transition is selected by the following switch:

```
switch (XSYMBOLNUMBER) {
{
  case 0:. start-transition
  nextstate;

  case 1: transition-1
  nextstate;
}
```

After pre-compiling it:

```
switch (_xSymbolNumber_ )
{
  .....
}
```

Start Transition

The start transition is included into the body of the generated yPAD function and has the same layout as transitions, with the following exceptions:

Assignment of initialization values to all local variables in the processes and procedures (if any) is executed. All DCL variables are filled with their default-values.

The start transition is selected by the special case-value zero in the switch-statement of the yPAD function.

> **Note:**
>
> FPARS in dynamic process creation are not contained in this version of the Cmicro Package.

Transitions

The transitions are translated in the order they are found and are only translated to the sequence of actions they consist of. The translation of actions are discussed in the subsection "Translation of Actions" on page 3342 following a few lines below.

PAD-End-Section

Each yPAD function is finished with:

```
END_PAD (yPAD_ProcessName);
```

The main reason for this is to make it possible to integrate other real-time operating systems.

> **Note:**
>
> Some compilers produce a warning if there is no return at the end of the yPAD function. Other compilers produce a warning "unreachable code", if there is a return at the end of the yPAD function. For this reason, a function returning macro `END_PAD` exists which can be expanded in accordance with the particular compiler used.

Translation of Actions

Translation of Output

SDL output statements are translated to the following basic structure:

- allocate the data area for the parameters of the signal to be output
- assign signal parameters
- send the signal, parameters will be copied
- release the data area for the parameters of the signal.

There are a lot of different output macros generated. The main reason for this is that for each output situation an optimized code is to be generated.

One differentiation is made for signals without parameters and signals with parameters. For a signal without parameters, suffix `_NPAR` is used for the macro generated and for a signal with parameters, suffix `_PAR` is used. The relevant output macro can then be expanded to a simpler output C function called `xmk_SendSimple`, if no signal priority is used.

Another differentiation is made for signals which are sent to the system's environment or which are sent internally in the SDL system. The suffix `_ENV` is appended to the macros which are shown here, if the signal should go to the system environment.

The different directives which can be used within the SDL suite to modify outputs are discussed in subsection "Modifying Outputs – Directive #EXTSIG, #ALT, #TRANSFER" on page 3325.

The other different output situations which are handled, will be described in the next subsections.

Output without TO and without VIA

If the user specifies output SignalName without TO and VIA in SDL, the Cmicro SDL to C Compiler calculates the receiver of the signal. It is also possible to have more than one receiver for the signal. During execution time, any possible receiver that are alive may be selected otherwise if no receiver can be found, the C function `ErrorHandler` will be called. The following code is generated:

```
ALLOC_SIGNAL_ppp(SignalNamewithoutPrefix,
                 SignalNamewithPrefix,
                 SignalParameterTypeStructureName)
```

ordinary assignment of Signal Parameters, if there are some...

```
SDL_OUTP_ppp(Priority,
             SignalNamewithoutPrefix,
             SignalNamewithPrefix,
             TO_PROCESS(ProcessNamewithoutPrefix,
                        ProcessNamewithPrefix),
             SignalParameterTypeStructureName,
             "SignalNamewithoutPrefix")
```

Note:

The `ppp` above stands for either `PAR` or `NPAR` for a Signal with or without parameters.

After expansion, the user will find a C function call to the `xmk_SendSimple` function or the `xmk_Send` function.

Priority is generated as `xDefaultPrioSignal` if no priority is specified for the signal with `#PRIO`.

`TO_PROCESS` is expanded to a function call if there is at minimum one (x, N) declaration in the system, where N is > 1. This function returns one of the possible receivers of the signal.

`TO_PROCESS` selects an active instance of the given process type It does not check for different types as receivers.

`TO_PROCESS` is expanded so that the pid is passed directly to one of the C functions `xmk_Send*`, if there are only (x,1) declarations in the system.

If the environment is the receiver of the signal, then the following code is generated:

```
ALLOC_SIGNAL_ppp(SignalNamewithoutPrefix,
                 SignalNamewithPrefix,
                 SignalParameterTypeStructureName)
```

ordinary assignment of Signal Parameters, if there are any...

```
SDL_OUTP_ppp_ENV(Priority,
                 SignalNamewithoutPrefix,
                 SignalNamewithPrefix,
                 ENV,
                 SignalParameterTypeStructureName,
       "SignalNamewithoutPrefix")
```

Note:

The `ppp` above stands for either `PAR` or `NPAR` for a Signal with or without parameters.

After expansion, the user will find that `ENV` is passed to one of the C functions `xmk_SendSimple` or `xmk_Send`. `ENV` is a special value used

inside the Cmicro Kernel to detect which signals are to be passed to the C function `xOutEnv`.

Output with TO clause

If the user specifies the output SignalName to pid in SDL, the Cmicro SDL to C Compiler generates the following code:

```
ALLOC_SIGNAL_ppp(SignalNamewithoutPrefix,
                 SignalNamewithPrefix,
                 SignalParameterTypeStructureName)
```

ordinary assignment of Signal Parameters, if there are some...

```
SDL_OUTP_ppp(Priority,
             SignalNamewithoutPrefix,
             SignalNamewithPrefix,
             pid-variable,
             SignalParameterTypeStructureName,
             "SignalNamewithoutPrefix")
```

Note:

The `ppp` above either stands for `PAR` or `NPAR` for a Signal with or without parameters.

Expansion reveals a C function call to the `xmk_SendSimple` function or the `xmk_Send` function.

Priority is generated as `xDefaultPrioSignal`, if no priority is specified for the signal with #PRIO.

Possible generated values for pid variable are `SDL_SENDER`, `SDL_PARENT`, `SDL_OFFSPRING` and `SDL_SELF` or an SDL pid variable. These values are passed to the `xmk_Send*` functions. The name of a process as specified in SDL may also be given.

Output with VIA clause

The Cmicro SDL to C Compiler computes the possible receivers in an output with the **VIA** clause. If there are several possible receivers, an error message is produced.

If there is exactly one receiver, the same code is generated as for SDL output without **to**.

List of Generated Output Macros

- `ALLOC_SIGNAL_NPAR`
  Allocating memory for signal without parameters
- `ALLOC_SIGNAL_PAR`
  same for signals with parameters
- `TO_PROCESS`
  Macro used to evaluate a receiver process instance, if necessary in the case of (x, N) declarations, where N > 1.
- `SDL_OUTP_NPAR`
  Output internally in the SDL system for signal without parameters
- `SDL_OUTP_PAR`
  same for signal with parameters
- `SDL_OUTP_NPAR_ENV`
  Output to the system environment for signal without parameters
- `SDL_OUTP_PAR_ENV`
  same for signal with parameters
- `SDL_ALTOUTP_NPAR`
  #ALT for an output internally in the SDL system for signal without parameters
- `SDL_ALTOUTP_PAR`
  same for signal with parameters
- `SDL_ALTOUTP_NPAR_ENV`
  #ALT for an output to the system environment for signal without parameters
- `SDL_ALTOUTP_PAR_ENV`
  same for signal with parameters
- `EXT_SignalName`
  if #EXTSIG is used in output
- `TRANSFER_SIGNAL`
  #TRANSFER is used in output

Translation of Create

The create action in SDL is translated to the following C code:

```
ALLOC_STARTUP_ppp(ProcessNamewithoutPrefix,
                  ProcessNamewithPrefix,
                  "ProcessNamewithoutPrefix, 0);
```

....assignment of start-up values (cannot be used in this version of the Cmicro Package)

```
SDL_CREATE(ProcessNamewithoutPrefix,
           ProcessNamewithPrefix,
           "ProcessNamewithoutPrefix, 0,
           VariableofCreatedProcess,
           PriorityofCreatedProcess,
           yPAD-functionNameofCreatedProcess);
```

`PriorityofCreatedProcess` is generated as `xDefaultPrioProcess`, if no priority is specified with #PRIO.

Translation of Set

The translation of set is restricted in a few areas in order to produce efficient code for a micro controller. For example, the SDL duration expressed by a real value in the context of timers is not implemented. The reason for this is that controllers do not have floating point operations or floating point operations are not used in order to increase the performance. For timers, such a high resolution is not necessary in most applications. The Cmicro Package uses a long value in its standard implementation to represent absolute time.

In order to make the examples below more readable, it is assumed that at least one timer with parameter is used in the system (macro `XMK_USED_TIMER_WITH_PARAMS` is defined in the generated file `sdl_cfg.h`). If the macro is not defined, then the handling for timers with parameters is not included.

Example 565 ----------

If the following is specified in SDL/PR:

```
Timer TimerName;
        ......
Set (now + durationvalue, TimerName) ;
```

or

```
Set (now + 22222, TimerName) ;
```

then the following code is generated:

```
SDL_SET_DUR  \
(xPlus_SDL_Time(SDL_NOW,SDL_DURATION_LIT(22222.0,22222,0)),
           SDL_DURATION_LIT(22222.0, 22222, 0),
           TimerName,
           TimerNamewithPrefix,
           yTim_timer2,
           "TimerNamewithoutPrefix")
```

Example 566 ----------

If the following is specified in SDL/PR:

```
Timer TimerName := TimerGroundValue ;---> see Note: on page 3348!
```

then the following code is generated:

```
SDL_SET_TICKS
(xPlus_SDL_Time(SDL_NOW, TICKS(SDL_INTEGER_LIT(22222))),
              TICKS(SDL_INTEGER_LIT(22222)),
              TimerName,
              TimerNamewithPrefix,
              yTim_timer2,
              "TimerNamewithoutPrefix")
```

The code after expansion then contains a function call to

```
xmk_TimerSet (TIMEEXPR,TimerNamewithPrefix,0).
```

`TIMEEXPR` is the result of the evaluation of **now** plus duration value.

```
#define SDL_SET_DUR(TIME_EXPR, DUR_EXPR, TIMER_NAME,
        TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING) \
xmk_TimerSet(TIME_EXPR, TIMER_IDNODE,0);

#define SDL_SET_TICKS(TIME_EXPR, DUR_EXPR, TIMER_NAME,
        TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING) \
xmk_TimerSet(TIME_EXPR,TIMER_IDNODE,0);
```

Example 567 ----------

If a timer with parameter is defined in SDL/PR:

```
Timer TimerName (integer);
...
set (now+1, TimerName (4711));
```

then the following code is generated:

```
SDL_SET_DUR_WITH_1IPARA(xPlus_SDL_Time(SDL_NOW,
                       SDL_DURATION_LIT(1.0, 1, 0)),
                       SDL_DURATION_LIT(1.0, 1, 0), TimerName,
                       TimerNamewithPrefix,
                       yPDef_z262_twp1,
                       yTim_TimerName,
                       "TimerName",
                       SDL_INTEGER_LIT(4711))
```

The code after expansion then contains a function call to

```
xmk_TimerSet (TIMEEXPR,TimerNamewithPrefix,4711).
```

Restrictions in the Use of Timers

- Timers with parameters are restrictively supported in Cmicro. There might be only one parameter of sort “integer”. This implementation has been chosen to achieve the highest efficiency.
- Duration values as real values are not supported in this version of the Cmicro Package, i.e. this:

  ```
  set (now + 5.5, TimerName)
  ```

  is **not allowed** (the real part is discarded i.e. 5.5 (= 5).

Translation of Reset

If the user specifies in SDL/PR:

```
Reset (TimerName) ;
```

then the following code is generated:

```
SDL_RESET(TimerNamewithoutPrefix,
          TimerNamewithPrefix,
          yTim_TimerName)
```

The code after expansion contains a function call to `xmk_TimerReset` (TimerNamewithPrefix).

For a timer with one integer parameter, the following macro call is generated:

```
SDL_RESET_WITH_1IPARA(TimerNamewithoutPrefix,
                      TimerNamewithPrefix,
                      TimerParStruct,
                      yTim_TimerName,
                      TimerValue)
```

Note:

Timers with parameters are supported with the restriction that only one integer parameter is allowed.

Translation of Call

As SDL procedures are implemented with the restrictions explained within subsection “SDL Restrictions” on page 3358, the following explanatory C code (to a procedure called `ex_proc`) is generated:

```
ex_proc (....C parameters ...);
```

All necessary parameters are routed via the C function call stack.

Translation of Call to a Procedure Returning Value / Operator Diagram

Operator diagrams and procedures returning values are – considering the call – handled in the same way please see the following explanatory example:

Example 568: Procedure Call ––––––––––––––––––––––––––––––

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

is translated to something like:

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

––

> **Note:**
> The value of returning procedure calls are transformed to C functions returning values.

Translation of Nextstate

The nextstate operation is generated at the end of each transition contained in the yPAD function, as follows:

- If the process performs simple nextstate operation:

  ```
  SDL_NEXTSTATE(State1, z000_State1, "State1")
  ```

  after preprocessing:

  ```
  return (z000_State1);
  ```

- If it performs a nextstate, which is defined as a dash state:

  ```
  SDL_DASH_NEXTSTATE
  ```

  which is defined as:

  ```
  return (XDASHSTATE);
  ```

Translation of Stop

A stop action is translated to:

```
SDL_STOP
```

which is defined as

```
return (XDORMANT);
```

which is good code saving. The Cmicro Kernel then enters the new state value into the Process State Table.

> **Note:**
>
> This table contains ordinary SDL state values as well as the values XSTARTUP and XDORMANT. XSTARTUP is generated for each instance which is to be statically created (in (x, N) declarations, where x is > 0). XDORMANT is the value which is used to tag a process instance as sleeping. In the case of creation this instance can be reused.

Translation of Return

```
#ifdef XFREEVARS
  FREE_PROCESS_VARS ()
#endif

SDL_RETURN
```

The macro definitions are:

```
#define SDL_RETURN \
        if (_xxptr != (unsigned char*) NULL) \
        { \
          XMK_MEM_FREE ((unsigned char *)_xxptr); \
        } \
        return ;
```

where xxptr is the pointer to the procedure instance data, as given via the C function call parameter list. Note, that the memory previously allocated directly before the procedure call is freed at the end of the procedure, not outside of the procedure.

Translation of SDL Expressions

In this section some of the translation rules for expressions are described. For more information see *"Translation of Sorts" on page 2595 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler* where for example the translation rules for literals and operators in the predefined abstract data types are given.

Now

SDL now is translated to the macro `SDL_NOW` which is expanded to the C function `xmk_NOW`. This function is exported by the module `mk_stim.c`.

Self, Parent, Offspring, Sender

The definitions for **self**, **parent**, **offspring**, **sender** are:

```
#ifdef  XMK_USED_SELF
#define SDL_SELF               xRunPID
#endif

#ifdef  XMK_USED_PARENT
#define SDL_PARENT             pRunPIDTable->Parent
#endif

#ifdef  XMK_USED_OFFSPRING
#define SDL_OFFSPRING          pRunPIDTable->Offspring
#endif

#ifdef  XMK_USED_SENDER
#define SDL_SENDER             P_MESSAGE->send
#endif
```

All the variables above are of type `xPID`. All variables are maintained by the Cmicro Kernel. `xRunPID` is a global variable which contains the pid of the SDL process which is currently running. `P_MESSAGE` is a pointer to the signal instance which is currently worked on.

Timer Active

An SDL timer active expression is translated to:

```
SDL_ACTIVE(TimerName, TimerName,
  yTim_TimerName)
```

which is expanded to:

```
xmk_TimerActive(TimerName)
```

A conditional expression in SDL is translated to a conditional expression in C.

Init Function

An explicit initialization function is not generated by the Cmicro SDL to C Compiler in any case.

The structure of the SDL system is not generated into the C code. What is seen in the generated code, is the behavior of the SDL system. Vari-

ables of processes are initialized during the start transition of a process and no information about the structure of the SDL system is available during run-time in the generated code.

An initialization function is generated only in that case if synonyms are used within SDL, which require an initialized C variable.

All this results in a more compact executable.

For example, the following use of an SDL synonym results in a generated initialization function:

```
synonym a integer := /*#CODE anyUserFunction () */
```

The following C code is then generated within the C function `yInit`:

```
yAssF_SDL_Integer(a, anyUserFunction (), XASS);
```

`yInit` is called by the Cmicro Kernel if the define

```
XMK_USED_INITFUNC
```

is generated into the file `sdl_cfg.h`, which is done in the case above.

Initialization of Synonyms

The Cmicro SDL to C Compiler allows SDL synonyms to be implemented as C macros and C variables.

Initialization is implemented within the C function `yInit` which is conditionally compiled.

Function main

The C function `main` is not automatically generated by the Cmicro SDL to C Compiler. This is unnecessary because the `main` function usually is provided from the user or the predefined `main` function can be used. Instead of an automatically generated `main` function, the user must supply the function body of `main`, for target applications. Guidelines can be found in the subsection "Implementation of Main Function" on page 3437 in chapter 67, *The Cmicro Library*.

Symbol Table File

The structure of an SDL system can be represented by a tree diagram. In SDL the root of the tree is represented by the SDL system followed by blocks, block substructures, processes and procedures[1]. Channels, channel substructures and signal routes are also represented in the tree. This tree is static, which means it cannot be modified during the run-time of an SDL system.

The SDL to C Compiler generates code so that this static structure is present in the generated code. This is good for debugging purposes.

The Cmicro SDL to C Compiler generates code so that this static structure is **not** present in the generated code, in order to spare memory. To enable debugging of the generated code, C comments are generated. Please consult the subsection "GR References" on page 3338.

A symbol table is necessary for the SDL Target Tester running on the host or the development system.

For more information consult chapter 68, *The SDL Target Tester*.

1. In SDL-92 several SDL systems can exist in parallel.

Generation of Identifiers

Processes and Process IDs (PID)

In order to implement the environment functions, it is important to notice that process IDs in Cmicro are generated into the `sdl_cfg.h` file, which must be included by the user's environment C module. These IDs are coded like it is described in "Generated Configuration File" on page 3302. More explanations are given in the following.

Since process IDs might become ambiguous, especially in block type and process type instantiations in SDL'92, the names of process IDs that are to be used in the environment functions are to be given a prefix. Using this prefix within the environment functions (`xInEnv`), it can be guaranteed that different process IDs (equates to "instance sets" in SDL'92) with the same name can be distinguished, which is necessary in order to send signals to the right process instance within the SDL system. On the other hand, prefixes are not necessary when all the process instance sets within the system have a different name. The Cmicro SDL to C Compiler uses an algorithm to calculate the prefixes in the most convenient way.

For example, if a process named "myprocess" exists only once within the SDL system, there will be no automatic prefix generated, e.g. the full process ID is

```
#define XPTID_myprocess 0
```

If, as another example, the process "myprocess" exists twice, for example once within a block called "myfirstblock" and once more within a block called "mysecondblock", the Cmicro SDL to C Compiler then creates two definitions which guarantee that the processes can be distinguished:

```
#define XPTID_myfirstblock_myprocess  0
#define XPTID_mysecondblock_myprocess 1
```

In this way, by adding scope names (block names), prefixes are always generated in a way so that no naming conflicts occur. Of course, for process and block type instantiations, the name of the instance is being used to generate this unambiguous prefix.

SDL process types (process instance sets in SDL'92), as well as SDL process instances are numbered consecutively beginning with zero. The

ordering of these numbers is the same as the ordering of the processes in the SDL/PR file.

The values 250 to 255 are reserved for internal purposes and must not be used for process type numbering. The Targeting Expert checks this rule automatically. For small systems this does not create any problems.

The Cmicro Kernel assumes the above definitions.

In the generated C code, the SDL values self, sender, parent and offspring, and variables of this type are represented by the `typedef xPID`. The intention is to have unique numbering of processes and their instances in the whole SDL system. This becomes necessary because of the Cmicro Code no longer containing the structure of the SDL system (system, block...). The `typedef xPID` is defined as

- `unsigned char` or `unsigned int`
  if there are only (x,1) declarations in the system no distinction between instances is necessary. This is automatically detected. See the flag `XMK_USED_ONLY_X_1` in the section "Automatic Scaling Included in Cmicro" on page 3426 in chapter 67, *The Cmicro Library*.

- `unsigned int` or `unsigned long`
  if there is at minimum one (x, N) declaration in the system, where N > 1, instances need to be distinguishable from each other.

There are a few macros defined to extract the process type number or the process instance number from a variable of the type `xPID` and to build an `xPID` variable from a process type number and a process instance number, the users do not have to think about the internal representation:

Example 569: Macros to extract process type or instance number ––

```
processtype     = EPIDTYPE(xPID_variable)
processinstance = EPIDINST(xPID_variable)
xPID_variable   = GLOBALPID(processtype, processinstance)
```

––

Signals and Timers

SDL signals and timers are numbered automatically by the Cmicro SDL to C Compiler so that they have a unique number over the complete system. Timers are represented by the values 1, 2, 3.... MT to the last timer of the MT timers in the system. After that follow ordinary SDL signal numbers beginning with MT+1, MT+2, MT+3... MT + MS.

When using the standard Cmicro Package, as delivered, then the values 0 and 251 to 255 are reserved for internal purposes. If the upper limit of 250 signals and timers is being reached, then the signal ID type has to be changed from `unsigned char` to `unsigned int`, thus allowing more than 60000 signals/timers to be handled. All these changes will be done if the flag XMK_USE_MORE_THAN_250_SIGNALS is set.

Caution!

The Cmicro SDL to C Compiler does not check for the upper limit of 250 signals being reached for a generated SDL system. Instead the Targeting Expert will check the amount of signals and timers in the SDL system and will inform the user.

Example 570: ----------

C code generated for signals and timers:

```
#define znnn_SignalName 1
#define znnn_SignalName 2
```

Where `znnn_` is the automatically generated prefix which is required to cope with the SDL scope rules. Remember, that processes in SDL can have the same name as signals, states etc. Prefixing, however, ensures uniquely named SDL objects in the generated C Code.

Example 571: ––

A system with 2 signals S1 and S2, and a timer TIMER1:

```
#define z049_TIMER1     1
#define z050_S1         2
#define z051_S2         3
```

––

When it comes to connecting the environment to the SDL system, the automatic numbering of signal IDs and timer IDs may not be required. If the user wants to prevent the automatic numbering of signals, then it is possible to #include a file containing all the signal and timer numbers. The file may contain something like:

```
#undef  SignalOrTimerName
#define SignalOrTimerName AnyValueAccordingToKernelRules
```

States

SDL states are consecutively numbered from 1 through to N for each process type. The values 0, and 250 to 255 are reserved for internal purposes in the Cmicro Package. This restriction incurs no foreseeable difficulty as processes should never have more than 50 States as a recommendation.

If there are even more states per process the flag XMK_USE_HUGE_TRANSITIONTABLES must be set.

The following C code generation is supplied for the header-section of the generated C file(s).

For each SDL process:

```
#define znnn_State1Name 1
#define znnn_State2Name 2
....
#define znnn_State3Name 3
```

Example 572: ––

For a process with 2 states S1 and S2:

```
#define z020_S1
#define z021_S2
```

––

These values are used in the state-index-table and in the generated C functions, wherever a nextstate is referenced.

SDL Restrictions

General

The Cmicro SDL to C Compiler handles SDL concepts according to the definition of SDL-92. In addition to the restrictions of all the SDL to C Compilers, the following additional restrictions are introduced for the Cmicro SDL to C Compiler:

- Inheritance of procedures
- Procedures with states
- Remote Procedure Calls
- Nested procedure call data scope
- Export / Import
- View / Reveal
- Enabling condition / Continuous signal
- Service and priority input and output
- Channel substructure
- Declaring an infinite number of process instances (x,) or (,)
- FPARS when creating a process
- Omission of parameters in a signal input
- Output via all
- Timers duration values cannot be real
- Timers with more than one parameter
- Timers with another parameter than sort integer
- The any expression
- Only the list of ADT and packages that are explained in the subsection "Exceptions for SDL Predefined Types" on page 3314 and the subsection "Exceptions for Implementations of Operators" on page 3317 are handled correctly with Cmicro.

The following restrictions are additional regarding the packages that are delivered together with the SDL suite.

sdth2sdl

It is impossible to read in header files created with Cadvanced and use them in Cmicro and the other way around. The reason is that it is impossible to mix up C code between Cadvanced/Cbasic and Cmicro.

Combining Cadvanced / Cmicro C Code

Mixing C code from different C Code Generators is not possible as the different code generators use their own run-time model and run-time data structures. Trying to mix up the C code will lead to compilation errors.

Light and Tight Integrations

The Light and Tight Integrations delivered with the SDL suite are only available for Cadvanced but not applicable to Cmicro. There are light and tight integrations for Cmicro but these are not part of the product.

Restrictions in Combination with SDL Target Tester

Scope Rules / Qualifiers

If the SDL Target Tester is to be used, then the **scope rules** of SDL are handled in a restrictive fashion. No information is generated for the system, block, block substructure, channel and signalroute. After applying the Cmicro SDL to C Compiler, all the structuring information is lost.

This means that it is impossible to address two different processes with the same name in different blocks. In order to avoid problems, give all processes, signals and timers in the system a different (unique) name.

Predefined Sorts

The predefined sort charstring and all the predefined sort that are based on the implementation of charstring (like predefined sorts from ASN.1) cannot be handled, if the SDL Target Tester is to be used. All predefined sorts which are generated into pointers in C cannot be used. In order to get a detailed description, please see in chapter 68, *The SDL Target Tester*.

Analyzer Restrictions

The restrictions in the SDL Analyzer, which also affects the Cmicro SDL to C Compiler, are summarized in *chapter 55, The SDL Analyzer*.

Chapter 66 The Cmicro SDL to C Compiler