Chapter

67 *The Cmicro Library*

The Cmicro Library consists of a configurable SDL kernel together with all the necessary SDL data handling functions. The collection of C functions and C modules make up the so called SDL machine.

The Cmicro Library is used in combination with the C code generated by the Cmicro SDL to C Compiler. For information on the generator see chapter 66, *The Cmicro SDL to C Compiler*.

The scaling facilities in Cmicro mean that it is useful for both micro controller and real-time applications. The Targeting Expert will help to scale and configure the generated code and the library. Please view chapter 60, *The Targeting Expert*.

The SDL Target Tester offers the ability to target tests and debug Cmicro code. Please see chapter 68, *The SDL Target Tester*.

Introduction

The Cmicro Library is required for handling the SDL objects that have been generated from SDL into C with the Cmicro SDL to C Compiler. This means that the C code which was generated with the Cmicro SDL to C Compiler cannot be used without the Cmicro Library. It also means that the C code generated with Cadvanced cannot be used together with the Cmicro Library and vice versa.

The Cmicro Library is a collection of C functions and C modules which consists of:

- The non preemptive Cmicro Kernel
- The preemptive Cmicro Kernel
- All functions which are necessary to handle SDL data
- Utility functions

Note:

The Cmicro preemptive kernel is only available if the according license is available.

Furthermore the Cmicro Library can be instrumented with SDL Target Tester functionality.

Before the Cmicro Library can be used, some adaptations and configurations must be made. The Targeting Expert is a tool which helps in configuring the application, node, component and the SDL Target Tester.

In this chapter, the C source code of the Cmicro Library and the generated C code of an application are described. Although a separate chapter is dedicated to the SDL Target Tester (see chapter 68, *The SDL Target Tester*), a few features are outlined here. The following topics are discussed:

1. The section "Differences between Cmicro and Cadvanced" on page 3364 helps in taking care of compatibility between different C code generators. There are things which must be observed in SDL diagrams.

2. The section "The SDL Scheduler Concepts" on page 3368 gives information about how the SDL scheduler works. This is also of interest for creating SDL specifications with the most highest conformity because the different SDL schedulers work differently.

3. The section "Targeting using the Cmicro Package" on page 3389 presents all the steps that must be carried out for doing targeting with Cmicro.

4. The section "Compilation Flags" on page 3394 presents a complete list of all the flags that the Targeting Expert recognizes. The different flags are explained in detail.

5. The section "Adaptation to Compilers" on page 3430 must usually be read before the C code generated by the Cmicro SDL to C Compiler and Cmicro Library can be compiled to form an executable. This section outlines the steps that must be carried out for introducing a C compiler which is not in the available list. The Targeting Expert offers an easy to use feature to add a new compiler. Please view "Compiler Definition for Compilation" on page 2836 in chapter 60, *The Targeting Expert*.

6. The section "Bare Integration" on page 3437 gives detailed information on how to adapt the generated SDL system to the environment. Communication with handwritten C code, as well as communication between SDL and the operating system or a naked machine, is described here.

7. The section "File Structure" on page 3472 explains the functionality of the different C files delivered with Cmicro.

8. The section "Functions of the Basic Cmicro Kernel" on page 3477 contains a list of C functions that are usually included in a target executable plus a short description.

9. The section "Functions of the Expanded Cmicro Kernel" on page 3499 contains a list of C functions that are usually not included in a target executable. The functions listed in this section can usually be left out because they have meaning for special forms of integration in target systems only. The functions include a short description.

10. The section "Technical Details for Memory Estimations" on page 3505 presents some information to the user which enables him to roughly estimate the consumption of memory in the target system. A self defined benchmark test is included, too.

Pay extra attention to the subsection "Automatic Scaling Included in Cmicro" on page 3426, which contains important information on the differences to the Cadvanced Library.

If examples of generated code are given, they are always shown in ANSI style. In addition, only the important parts of the code are shown to increase readability in the examples.

Differences between Cmicro and Cadvanced

General

This description deals with the differences between the generated C code of the Cadvanced SDL to C Compiler and the Cmicro SDL to C Compiler and the run-time libraries. There are differences because the main application area for the code generators differ.

Some of the differences discussed in the following are of interest for SDL users, while others are not.

SDL Restrictions

The Cmicro SDL to C Compiler has more SDL restrictions than the Cadvanced SDL to C Compiler. The additional restrictions are described in the subsection "SDL Restrictions" on page 3358 in chapter 66, *The Cmicro SDL to C Compiler*.

Furthermore, there are restrictions within the Cmicro Library that may affect the user's SDL system design. These restrictions are listed in the following. More technical information can be found in the section "Generation of Identifiers" on page 3354 in chapter 66, *The Cmicro SDL to C Compiler*.

- The run time model in Cmicro is such that there are global variables used in the generated C code and the Cmicro Library. The code generation uses "Auto initialization in C" quite extensively because the C compiler then can produce code that is more efficient than if initialization would take place within a generated C function (like `yInit`).
- The run-time model of the preemptive kernel requires function recursion from the C compiler.

Scheduling

First, it must be emphasized that both schedulers conform fully with SDL, although the schedulers introduce a different behavior.

Cadvanced uses a process ready queue together with signal queues in order to schedule processes. Cmicro does not use a process ready queue but all scheduling is derived from one physical queue. This physical queue represents all SDL process input ports. Logically, or seen from SDL, each SDL process has its own input port.

Another difference is the preemptive scheduler of Cmicro, if it is used.

Thus, different execution of processes will occur.

It is also a question of SDL design whether the differences between Cadvanced and Cmicro can be externally recognized.

If no SDL process assumes a particular real-time behavior from its communicating partner process then the behavior – as seen from a black box view – is always the required one.

Generation of Files

The Cmicro SDL to C Compiler generates some more files. This is of interest when implementing build scripts, makefiles and so on.

It is important that after each change in the SDL system the Cmicro Library is re-compiled. The reason for this is the automatic scaling facility. Please view "Automatic Scaling Included in Cmicro" on page 3426.

Environment Handling Functions

The main differences arise when considering the environment handling functions. However, this only has consequences if the SDL environment is the same in both cases (if Cadvanced is used for simulation only and Cmicro for targeting, then the environment functions are to be implemented twice in any case).

Instead of including `scttypes.h` as in Cadvanced, for Cmicro the following include statements are to be introduced in the header of the environment module:

```
#include "ml_typ.h"
#include "<systemname>.ifc"
```

In general (for Cmicro as well as Cadvanced) four C functions are used to represent the SDL environment, namely

- `xInitEnv`,
- `xInEnv`,
- `xOutEnv`,
- `xCloseEnv`.

The functions have the same general meaning for Cadvanced and Cmicro, but there are a few differences so that it is necessary to implement the environment twice.

For Cmicro, each of the above C functions is compiled only if it is required (selected by `XMK_USE_xInitEnv, XMK_USE_xInEnv`...)

Differences occur in the declaration of the C functions `xInEnv()` and `xOutEnv()`.

The `xInEnv()` function of Cmicro carries no parameters.

The `xOutEnv`() function of Cmicro carries a few parameters which represent the signal which is to be output to the environment, including the signal parameters. It is recommended that the definition of the C function `xOutEnv()` should be written twice, once for Cadvanced and once for Cmicro.

The environment functions can also be generated with the help of the Targeting Expert.

Another difference is that signals and processes are identified in different ways. Cmicro does not use identifications like `xIdNode`. Instead, it uses fixed C defines to identify signals and processes. This is beneficial in reducing the amount of memory but has the consequence that each access to any signal and any process is to be implemented differently. Process type IDs are generated into the file `sdl_cfg.h`, signal IDs and type definitions are generated into the `<systemname>.ifc` file. The chapter about the Cmicro SDL to C Compiler gives more details on how identifiers are generated and can be used.

Including C Code in SDL by User

C code may be included in SDL by the user in the following cases:

- In order to connect SDL to the environment in a way other than via the C functions `xInEnv() / xOutEnv()`

- To use C constructs, which do not exist in SDL
- To use existing C code (e.g. C library functions)
- To implement the body of ADTs
- In an SDL task.

If any C code or C identifiers are used, then users must use the right identifiers and functions.

Generated C Code

Of course, the generated C code is different when comparing Cmicro with Cadvanced. It would take too much room to list all the details in this subsection. In any case these differences are of interest for certain technical reasons only and not for pure SDL users.

To compare the different code generator outputs, the user should refer to the subsection "Output of Code Generation" on page 3326 in chapter 66, *The Cmicro SDL to C Compiler*.

General Recommendations Regarding Compatibility

In order to reach full compatibility between Cadvanced and Cmicro, the following general recommendations should be followed:

- In general, C code should not be used in SDL diagrams.
- If there is no option other than to use C code, the C code should be written as compatible as possible, i.e. the C code should be written without using any C code generator or compiler specific commands. Also the C code should be placed at dedicated locations, and should not be distributed over the SDL diagrams.
- If it is not possible to reach full compatibility, then the C code must be written twice. A switch, which is used when invoking the C compiler, selects C code either for Cadvanced or for Cmicro. The macro `XSCT_CMICRO` will help to distinguish automatically.

The SDL Scheduler Concepts

In this section, the concepts of the Cmicro SDL scheduler are outlined, with particular emphasis on basic SDL, the handling of the queue, scheduling, signals, timers, states etc. To obtain information about data in SDL, especially ADTs, please consult chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*. Here both predefined and user defined ADTs are outlined.

Signals, Timers and Start-Up Signals

Data Structure for Signals and Timers

Each signal that is either an ordinary SDL signal, a timer or the start-up signal used for the dynamic process creation, is represented by three structures:

- A C structure defining the entries in the queue
- A C structure defining the header of each signal
- A generated C structure representing the parameters of the signal

The first and the second is defined in `ml_typ.h`, the third is defined in the generated code as `yPDef_SignalName`. Some structure components are conditionally compiled, which is used to scale the system. Please view the following C structure:

```
typedef struct
{
   #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
       xPID          rec;   /* Receiver process */
   #endif

   xmk_T_SIGNAL  signal;    /* Signalcode       */

   #ifdef XMK_USE_SIGNAL_PRIORITIES
      xmk_T_PRIO  prio;     /* Priority         */
   #endif

   #ifdef XMK_USE_SIGNAL_TIME_STAMP
      xmk_T_TIME time_stamp;/* Timestamp        */
   #endif

   #ifdef XMK_USE_SENDER_PID_IN_SIGNAL
       xPID         send    /* Sender process   */
   #endif
```

```
  #ifdef XMK_USED_SIGNAL_WITH_PARAMS
    xmk_T_MESS_LENGTH mess_length;
    union
    {
       void          *ParPtr;
       #if (XMK_MSG_BORDER_LEN > 0)
         unsigned char ParCopy[XMK_MSG_BORDER_LEN];
       #endif
    } ParUnion;
  #endif

} xmk_T_MESSAGE;
```

- `signal:`
  represents a C constant which uniquely identifies the signal or timer across the complete system.

- `rec and send:`
  represents the PID values of the receiver and sender of the signal. See the subsection "Processes and Process IDs (PID)" on page 3354 in chapter 66, *The Cmicro SDL to C Compiler* for details. These are conditionally compiled.

- `prio`:
  is used to store the priority of the signal. This is specified by using the directive `#PRIO` in the SDL diagram. It is compiled only, if signals with priority are stipulated by the user.

- `timestamp:`
  is used to store a timestamp, which is set when the SDL signal is put into the queue (output time). This component is also conditionally compiled.

- `mess_length:`
  represents the amount of parameter bytes in the signal.

Three different cases are to be considered concerning signals with parameters:

- If the signal has no parameters, then `mess_length` is set to 0 and `ParPtr` is a `NULL` pointer.

- If the signal parameters are larger than XMK_MSG_BORDER_LEN (*) bytes, then `mess_length` is set to the number of bytes and `ParPtr` is undefined.

- If more than XMK_MSG_BORDER_LEN (*) bytes of parameters are to be transferred, then `mess_length` defines the amount of bytes used for the parameters and `ParPtr` points to a dynamically allocated area.

Note:

This margin of XMK_MSG_BORDER_LEN bytes can of course be modified by the user to prevent dynamic memory allocation for any signal in the queue, or in contrast, to always use dynamic memory allocation. See the file `ml_mcf.h` to modify this.

The second structure, which encapsulates the above one, is used to administrate the signals in the queue, as required by the FIFO handling and SDL save construct:

```
typedef struct _T_E_SIGNAL
{
  xmk_T_MESSAGE       Signal;
  struct _T_E_SIGNAL *next;

#ifdef XMK_USED_SAVE
  xmk_T_STATE         SaveState;
#endif /* XMK_USED_SAVE */
} T_E_SIGNAL ;
```

- `next:`
  is a pointer which refers to the next entry of the queue. This is used by the Cmicro Kernel to get the next signal after the current one has been worked on.

- `SaveState:`
  contains either a dummy state value XEMPTYSTATEID or the state in which the process saved the signal. This is necessary to compare when the signal is to be consumed after state changes in the receiver process. It is used only if save is used anywhere in the SDL system.

As already mentioned, the above structure is used for ordinary SDL signals, timers and the start-up signal for the dynamic process creation.

No differentiation is made between signals and timers, except that signals and timers have a different identification (signal in the `xmk_T_SIGNAL` structure).

When a process is to be created, a start-up signal is sent. The start-up signal is tagged by a special priority value and a special id.

Dynamic Memory Allocation

Dynamic memory allocation is in principle not necessary with Cmicro. There are SDL systems which can live without any dynamic memory allocation and there are SDL systems that require dynamic memory allocation from the user's point of view. The users should in general try to prevent any dynamic memory allocation due to the problems this introduces. Soon or later memory leaks will occur.

Cmicro offers its own dynamic memory allocator. Please view "Dynamic Memory Allocation" on page 3450 for getting more information on this.

In the following subsections, the exceptions for when Cmicro uses dynamic memory allocation are listed.

Signals and Signal Parameters

In order to cope with efficiency, dynamic memory allocation should not be done, whenever possible. Cmicro offers two principles of memory allocation for signal instances, namely:

1. Signal instances are allocated from a static memory pool only.

   The static memory is allocated during compilation. The pool's size is predefined with the `XMK_MAX_SIGNALS` macro. To enable this configuration, the macro `XMK_USE_STATIC_QUEUE_ONLY` is to be set. This principle has the disadvantage that if there is no free memory in the static memory pool, a fatal error occurs. The user is however given the possibility to react on this error situation because the `ErrorHandler` C function is called.

2. As another principle, it is possible to first take memory from the static memory pool. If no more memory is available in that pool, further signal instances are created from the dynamic memory pool.

> **Caution!**
>
> This configuration is not available when the preemptive scheduling mechanism is used.

To enable the configuration, the macro XMK_USE_STATIC_AND_DYNAMIC_QUEUE must be set. The static memory pool's size is still predefined with the XMK_MAX_SIGNALS macro.

When signal instances are allocated from the static memory pool, the allocation procedure is the most efficient.

When the second principle is used, the execution speed will of course dramatically slow down because dynamic memory allocation happens. This change in the behavior of the Cmicro Kernel may cause problems in a real-time environment and the user should keep this in mind.

The static memory pool above is implemented as a predefined array in C. The array is dimensioned with the XMK_MAX_SIGNALS macro.

Dynamic memory is requested with the xAlloc C function and released again with the xFree C function. The body of both these C functions must in any case be filled out appropriately by the user in all cases.

Signal parameters are to be allocated dynamically, if the amount of parameter bytes exceeds a predefined constant. If the amount of signal parameters is below or equal this predefined constant, the parameter bytes are put into the signal's header.

The default value for this predefined constant XMK_MSG_BORDER_LEN is 4 bytes. Dynamic memory allocation only occurs if a signal carries more than XMK_MSG_BORDER_LEN bytes, where the latter is defined as 4.

If the user defines XMK_MSG_BORDER_LEN as 0, then for each signal, which has parameters, the C function xAlloc() is called.

If the user defines XMK_MSG_BORDER_LEN to 127, then

- for signals which have 127 parameter bytes or less, the parameters are copied into the xmk_T_SIGNAL structure.
- for signals which have more than 127 parameter bytes, the parameters are copied into an allocated area (using the C function xAlloc()).

Due to this mechanism, the allocation of signal parameters is also very fast. Releasing the memory is performed automatically by calling the xFree C function.

Predefined Sorts

Some of the predefined sorts require dynamic memory allocation. The sorts are

- charstring
- any ASN.1 sort that is based on charstring
- predefined Generators like Array (without a specified upper limit), String, Powerset, Bag, Ref

SDL Target Tester

If the SDL Target Tester is used, there are some more allocations from the dynamic memory pool. The SDL Target Tester allocates one block of dynamic memory in the start phase. This block is never de-allocated again. The size of the block depends on the amount of process types in the system. Another dynamic memory allocation takes place when a binary frame is to be sent to the host machine. The blocks that are allocated here are de-allocated again after the frame is put into the physical transmitter buffer.

Overview for Output and Input of Signals

Output and input of signals is performed according to the rules of SDL. To get detailed information about the implementation of output and input, please consult the subsection "Output and Input of Signals" on page 3384.

Signal instances are sent using the C function `xmk_Send()` or `xmk_SendSimple()`. The function takes a signal and puts it into the input port of the receiving process instance.

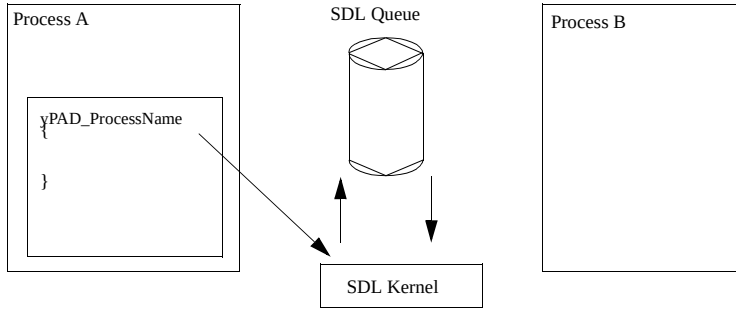

Figure 580: Queueing - sending side

There is no process ready queue. Physically, the queues of the different processes of the system are represented by just one queue. There are different principals to create signal instances which the user can choose between. The principals are explained within the subsection "Signals and Signal Parameters" on page 3371.

From an abstract point of view, it does not matter, if there is one physical queue for all the signal instances in the system, or if there is one physical queue for all the signal instances sent to one process instance. The fact that Cmicro uses one physical queue for all signals in the system only, has no effect on SDL users and conforms to the semantic rules of SDL. The scheduling simply depends on the ordering of signals in the queue. In the case of a preemptive Cmicro Kernel scaled this way, there is one linked list of signals per priority level all using the array mentioned above. See the subsection "Scheduling" on page 3377 for details.

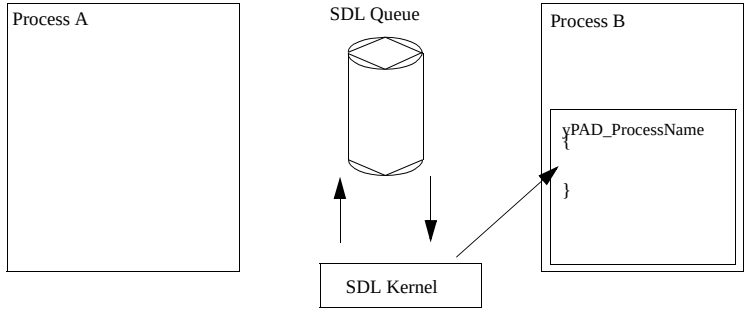

Figure 581: Queueing - receiving side

When working on a signal, the Cmicro Kernel decides between four different constellations:

- The receiving process instance is active. In its current state the signal is to be saved. The Cmicro Kernel tags the signal as "saved" and works on the next signal.
- The receiving process instance is active and there is a transition to be executed receiving the current signal. The Cmicro Kernel fires the transition.
- The receiving process instance is active but there is no transition to be executed for that signal in the current process state. The signal is in accordance with SDL rules implicitly consumed by the process.

- The receiving process instance is not active, i.e. either not yet created or already stopped. The signal will be discarded and the C function `ErrorHandler()`, discussed within the subsection "User Defined Actions for System Errors – the ErrorHandler" on page 3455, will be called.

In any case, except when being saved, the signal will be removed from the queue and returned to the list of free signals.

After performing the nextstate operation, the input port is scanned in accordance with the rules of SDL to find the next signal which would cause an implicit or explicit transition.

There is no specific input function. This functionality is contained in the Cmicro Kernel.

Timers and Operations on Timers

With the delivered timer model, all timer management entities fully conform to SDL. This means that more memory is required to implement this timer.

For each timer, there is a C structure defined in `ml_typ.h`.

```
typedef struct _TIMER
{
  struct _TIMER *next ;

  xmk_T_SIGNAL   Signal   ;
  xmk_T_TIME     time     ;
  xPID           OwnerProcessPID ;

} TIMER ;
```

`xmk_T_TIME` is defined as a long value.

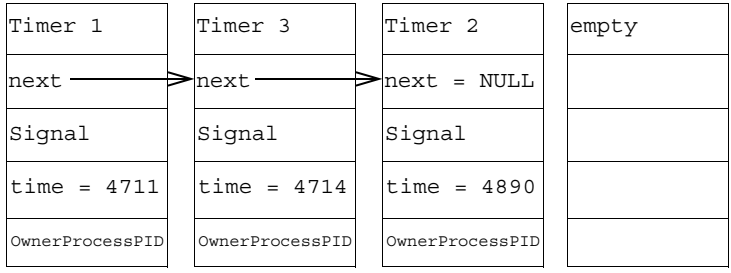

Figure 582: Handling of timers (timer model 1)

The component

- `next` is used to refer to the next entry;
- `signal` is the timer identification;
- `time` is used to store the absolute time when the timer should expire;
- `OwnerProcessPID` is used to refer to the process PID for which the timer was set.

As can be seen above, timers are implemented as a forward linked list.

- An array of timers is allocated during compile time
- The size of the array is calculated by the Cmicro SDL to C Compiler as the required maximum. It is defined by the macro MAX_SDL_TIMER_INSTS. This is also valid for multiple process instances, so that there is always enough memory to handle all timer instances which are possible. (If the user wishes to reduce the occupied RAM memory for timers, he must evaluate the maximum amount of timer instances required during run-time and modify the above define by hand).
- The timer which expires first is always put to the front of the linked list (increasing performance).
- The time value is stored as absolute time from the start of the SDL system.

Processes

Data Structure for Processes

Each process is represented by structures and tables containing SDL information, for example, tables containing transition definitions and a structure representing the variables of the process. The typedef for variables is generated in the generated code and named `yVDef_ProcessName`. For each process instance, there is one array element defined statically during compile time.

The structures and tables for processes are described in detail in the subsection "Tables for Processes" on page 3328 in chapter 66, *The Cmicro SDL to C Compiler*.

Scheduling

The Cmicro Kernel supports in principle the following scheduling policies:

- Non Preemptive Scheduling:
  Transitions in SDL cannot be interrupted.
- Preemptive Scheduling with Process Priorities:
  Transitions in SDL can be interrupted by any external or internal output. This could be a signal coming from the environment, i.e. an interface driver.
- In addition, signal priorities can be used to specify the ordering of signals in the signal queue. Signal priorities are discussed in each of the subsections mentioned above.

General Scheduling Rules

- All of the above policies basically operate on the SDL queue. Physically, the Cmicro Kernel uses one array for all process queues in the system. From the SDL point of view, the implementation conforms to SDL because each SDL process virtually has its own queue in the implementation.
- signal priorities can be used in all cases.
- process priorities are given priority treatment.

- If no signal priorities are assigned, then the ordering of signals depends on their arrival (FIFO strategy).
- In order to increase the performance, there is no process ready queue implemented. The scheduling of processes is fully derived from the SDL queue.

Further explanation of the scheduling is in the next subsection.

Non Preemptive Scheduling

The Cmicro Kernel takes the first signal from the queue and if the signal is not to be saved the appropriate SDL transition is executed until a nextstate operation is encountered. Outputs in SDL transitions as well as SDL create operations are represented by signals, where create signals are given priority treatment no matter whether signal priorities are used or not. This guarantees that there will not be any problems when a signal is sent to a process just before the process has been dynamically created.

The ordering of signals in the queue can be affected by using signal priorities (#PRIO directive for signals).

Whenever an output takes place, the Cmicro Kernel inserts the signal into the queue according to its priority. High priority signals are inserted at the queue's head, low priority signals at the queue's end. Create signals are still given priority treatment.

Note:

The priority of a create signal in the standard delivery is set to one.

This, as well as all the default signal priorities, is user-definable in the file `ml_mcf.h` (generated by the Targeting Expert). In this way it is possible to define signals with a higher priority then the create signal. The user should remember not to do so!

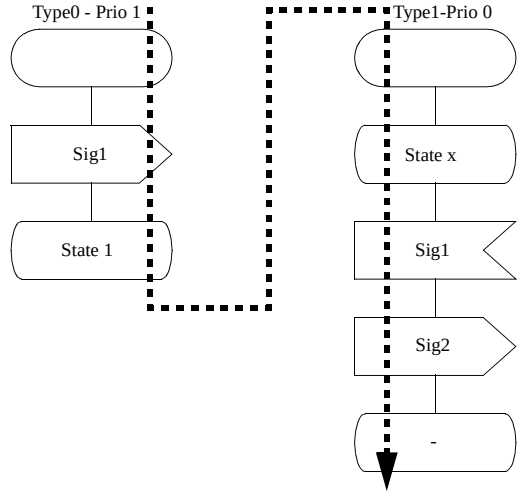

Figure 583: Non preemptive scheduling

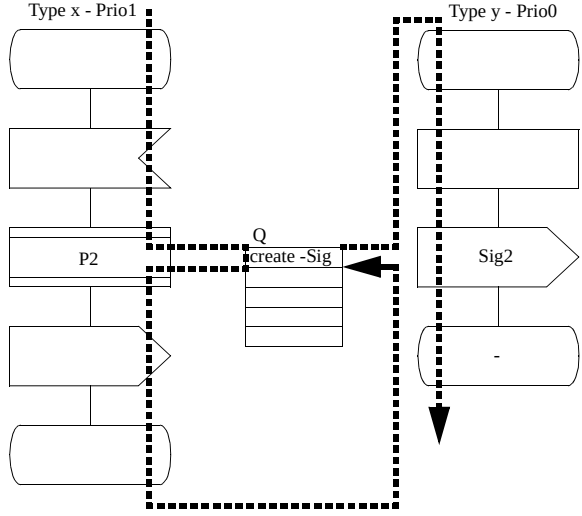

Figure 584: Scheduling for Create

A signal (this is either an ordinary SDL signal, a timer signal or the internal start-up signal in the event of SDL create) with the highest prior-

ity is always put in front of the SDL queue, a signal with the lowest priority is always put at the end of the SDL queue.

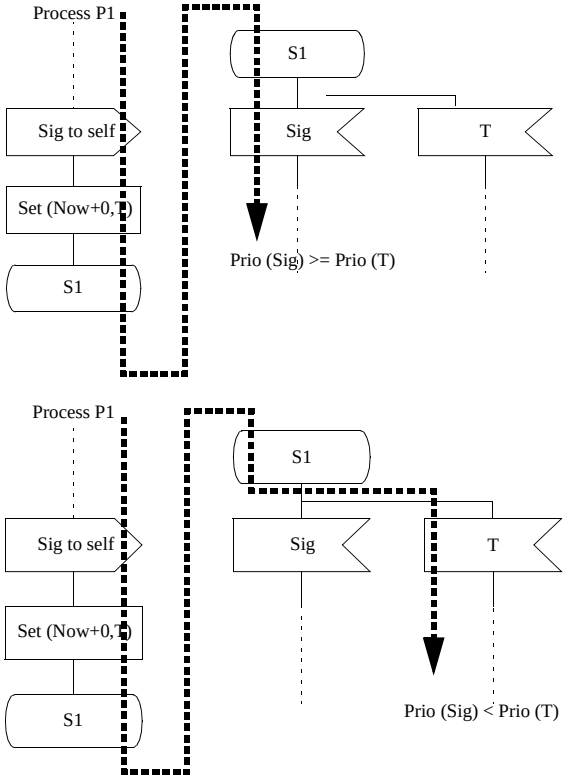

Figure 585: Signal priorities

For applications, which do not have time critical requirements, the non preemptive scheduling policy is the correct one to implement. The Cmicro Kernel memory requirements are also reduced when the non-preemptive scheduling policy is implemented. For example, for an interface with a very high transmission rate, the preemptive scheduling policy is better suited in order to increase the reaction time on external signals coming from the environment.

On starting the SDL system, processes are statically created according to their order of priority.

Preemptive Scheduling with Process Priorities

Note:

The Cmicro preemptive kernel is only available if the according license is available.

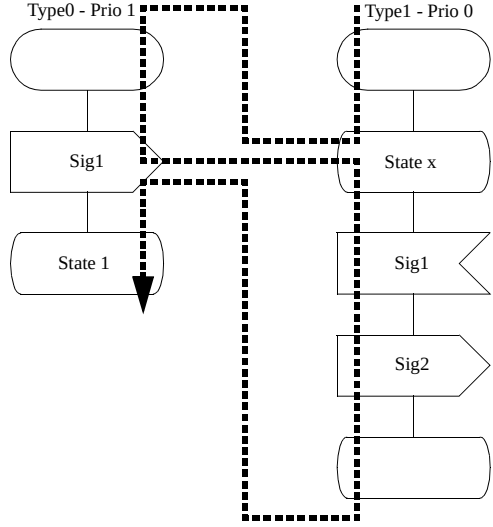

Figure 586: Preemptive scheduling

SDL assumes the start transitions of all statically created processes are already finished at system start-up (a transition takes no time according to SDL semantics). To simulate this, the Cmicro Kernel starts all statically created processes in the order of their priority before working on any signal. Preemption is disabled during the start-up phase of the system. There are two C functions namely `xmk_DisablePreemption` and `xmk_EnablePreemption` which can be used to prevent the Cmicro Kernel from performing a context switch. In this way it is possible to affect the scheduling from within the SDL system (using the #CODE directive).

The preemptive scheduling policy is absolutely necessary if an application consists of a mixture of processes, of which some have to react very quickly to external events, while others require enough time for pro-

cessing. The Cmicro Kernel does all things necessary to schedule such a mixture of processes.

Users only have to specify a high priority for processes which have to react after a short time. This can be done with the #PRIO directive for processes. If no `#PRIO` directive is specified, then a process is given the default value, which is specified by the user in the file `ml_mcf.h` (please view the subsection "Compilation Flags" on page 3394).

The highest priority is represented by the value zero. The numbering has to be consecutive with the priority decreasing with increasing numbers. Processes with the same priority are on the same priority level. The default process priority must be in the range of zero to the lowest priority value used in the system. Assume the following priority levels:

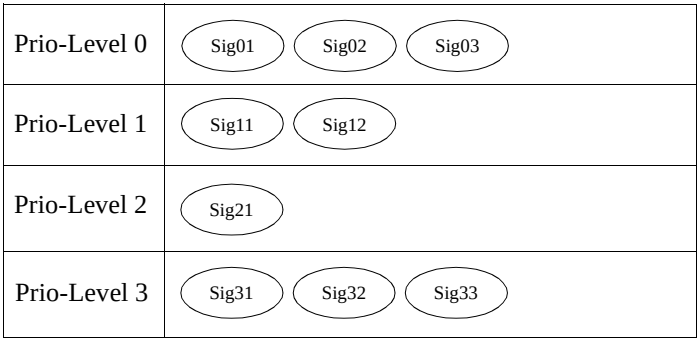

Figure 587: Priority levels

In the figure above there are signals queued for processes on four different priority levels. These would be worked on in this order:

According to their priorities the signals on priority level zero are consumed first, afterwards those of level one, two and last three. This is relevant only, if no signals are sent during the transitions executed because of the signals.

Rules

- At system start, SDL processes are statically created in accordance with their defined priority. During their start transitions it is possible for processes to send signals or create other processes but no preemption will take place until the start-up phase is completed, i.e. all

static processes have completed the transition from start state to first state.

- Processes running on a priority level never interrupt other processes running on the same priority level, thus two instances of the same type never run at one time.
- A higher priority level is entered, if a signal results in a process with a higher priority running.
- A lower priority level is entered, if all signals on the currently active priority level are consumed so that still no process is running on this priority level.
- Global variables are used to store some information concerning processes. The Cmicro Kernel is responsible for the storing and reloading of these, if a preemptive kernel is implemented.

Note:

If C variables are to be used in the SDL application, i.e. (x, N) declarations are used where N > 1 these variables are only available once and not once per process instance.

Restrictions

In order to produce portable C code, this version of the Cmicro Kernel uses recursive C function calls. A few C compilers available on the market do not support recursion. If such a C compiler is required, the user cannot use preemptive scheduling with process priorities.

Create and Stop Operations

No dynamic memory allocation occurs if an SDL create operation is performed. No freeing of memory occurs if an SDL stop operation is performed.

Data of a process instance is represented by the following typedef structs:

- A typedef struct representing PID values (parent and offspring) stored for an instance
- A typedef struct representing SDL states of an instance
- A typedef struct representing SDL data of an instance

The reason for using such a structure is to make it possible for the Cmicro Kernel to operate on the PID structure and the table with SDL states, so that no code needs to be generated in the application to update such variables.

An array of N elements for each of these typedef structs is generated, where N is the maximum number of process instances. The maximum number is to be specified in the process declaration header in the SDL diagram.

After performing a stop action, old PID values might exist in variables of other processes. The synchronization between processes to prevent situations where signals are sent to dead processes is left to the discretion of the user. If a process sends a signal to a non existent process, where nonexistent means either “never created” or “is dead”, the `ErrorHandler` is called and the signal is discarded (SDL conform).

When the Cmicro Kernel stops a process, the input queue assigned to the process stopped will be removed. No interpretation error occurs for the signals which existed in the queue before the process was stopped.

Note:

It is also possible for the user to implement dynamic memory allocation for process instance data. Some C defines are to be redefined in this case.

Output and Input of Signals

The actions to perform an output in the generated code are as follows:

- A struct variable is initialized with the type of the signal parameters, not using memory allocation.
- The struct variable is then filled, parameter per parameter in generated code.
- One of the `xmk_Send*` functions is called with a pointer to the location of the struct variable. There are several `xmk_Send*` functions in order to use the most effective one in SDL output.

Within the `xmk_Send*`- functions there are a few checks performed. For example, if the receiver is a NULL-PID, then the ErrorHandler is called.

Next, the environment C function `xOutEnv` is called. This function is to be filled out by the user. The function decides if the signal should be sent to the environment. The information necessary can be extracted from the signal ID, the priority or the receiver of the signal. If `xOutEnv` has "consumed" the signal, `xmk_Send*` returns immediately. `xOutEnv` has to copy the parameters of the signal to the receiver of the signal in the environment because after returning, the parameters will no longer exist.

If the signal is not environment bound, then the signal is sent to an internal SDL system process and `xmk_Send*` inserts the signal into the queue. This is done according to the priority of the signal (see subsection "Assigning Priorities – Directive #PRIO" on page 3323 in chapter 66, *The Cmicro SDL to C Compiler*.

If the signal carries no parameters, or if the signal parameters are represented by less than or equal to `XMK_MSG_BORDER_LEN` bytes, no dynamic memory allocation occurs and possible parameters are transferred directly by copying them into the signal header.

If more than `XMK_MSG_BORDER_LEN` bytes parameters are to be transferred dynamic memory allocation occurs. A pointer in the signal header then points to this allocated memory area. Freeing is done after consumption of the signal at the receiver process after executing the next-state operation or after the signal was consumed by the environment.

In order to implement this strategy, each signal carries a field "data length" in its header, to detect if a pointer is transferred or a copy of the parameters.

Note:

There are several possibilities to send signals to the environment by not using `xOutEnv`. The user should have a look at the `#EXTSIG`, `#ALT` and `#TRANSFER` directives which can be used in SDL Output. The one most similar to SDL, however, is the one which simply uses the C function `xOutEnv`.

At the receiver's side, when the input operation is to be performed, it is checked if the signal is to be saved or discarded. In the case of save, the next signal contained in the queue is checked and worked on. If on the other hand the signal is to be discarded, in the case of an "implicit transition", i.e. no definition is present to handle that signal in the current

state, the signal is then deleted from the input port of the process and, using the SDL Target Tester, the user is notified.

Otherwise, the signal leads to the execution of the transition. After performing the nextstate operation, the signal is deleted from the input port of the process and scheduling continues according to the rules of the scheduler again.

In the case of a so scaled preemptive Cmicro Kernel, the signal remains active in the input port until the nextstate operation is executed, although other processes can interrupt the running one (preemption).

Nextstate Operation

The nextstate operation is implemented by a return (return value) statement in the generated code. Several return values are possible to control the action to be taken by the Cmicro Kernel. The PAD function representing the SDL process can return any of the following (PAD means "Process Activity Description").

The return-value either expresses

- An ordinary SDL state change, the value is a C constant representing the SDL state as a simple integer (generated as #define),
- no SDL state change, indicated by the value `SDL_DASH_STATE`,
- SDL process stop operation, indicated by the value `SDL_STOP`.

After recognizing the action to be taken, the Cmicro Kernel either writes the process state variable or stops the SDL process.

The signal which was just being worked on is deleted from the input port. This includes the freeing of the memory area allocated for the signal, if memory was previously allocated.

Decision and Task Operations

No action is to be performed by the Cmicro Kernel for decisions and tasks, except those to trace these actions for the SDL Target Tester.

For SDL decisions and SDL tasks, C assignments and C function calls are implemented. Function calls are used in the case of an ADT or where simple C data operations (i.e. =, >, >=, ==) cannot represent the SDL operation wanted.

Note:

Function calls, however, are not necessarily generated, i.e. if the user defines C macros/defines instead of C functions for an ADT. Please consult the subsection "Abstract Data Types" on page 3314 in chapter 66, *The Cmicro SDL to C Compiler*.

Procedures

There are some special topics regarding procedures in SDL. The most important are:

- The location of data and how to access data
- Scope and visibility rules
- Procedures with states
- Recursivity
- Who can call a procedure

Note that procedures with states and remote procedure calls are **not supported** within the Cmicro Package.

It is possible to use global procedures (SDL'92) which makes it possible to specify a procedure once, by allowing several processes to call it.

Recursion is allowed, but should be introduced only if an algorithm cannot be designed alternatively. In most cases, algorithms and recursivity are subjects for an ADT.

Procedures returning values are also implemented. The return value in SDL is mapped to a return value in C. Procedures not returning values are mapped to C functions returning void.

The remaining part is the location of procedure data and the access to procedure data. Here another restriction exists, namely that it is not possible to access data of the father procedure of a procedure without declaring this explicitly.

Data, which belongs to a process is always located as a global array in C (for x, N process declarations, where N is >1). Data which belongs to a procedure is always allocated on a C stack for the called procedure.

Procedure Calls

An SDL procedure call is implemented as a direct function call in C. Each formal parameter is passed as a C parameter to the function. Ac-

cess to global data of the calling process is possible only if the procedure is not a global procedure. The C code generated for a local procedure uses global C variables of the surrounding process.

Data which is declared locally within a procedure is also allocated on the C stack.

No dynamic memory allocation is performed as procedures with states are not handled.

The following example shows the mapping for procedures returning values.

Example 573: 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 returning values.

An SDL procedure can be called more than once. No conflicts occur if using a preemptive Cmicro Kernel.

Procedure Body

For each SDL procedure, there is one C function generated. The Procedure body can contain the same SDL actions as the process body. The same code generation is performed with the exception of a few statements declaring temporary variables.

Within global procedures, no objects of the calling process can be used without declaring them via formal parameters. Another restriction is that each output in a global procedure must be specified with **to PID**.

Blocks, Channels and Signal Routes

No C code is generated for blocks, channels and signal routes, except the C comment, which tells the user the location of processes and procedures.

Targeting using the Cmicro Package

Directory Structure

At first it is necessary to know the directory structure the Cmicro Package is stored in.

On UNIX, the Cmicro Package is contained in `$sdtdir/cmicro`.

In Windows, the Cmicro Package is contained in `%SDTDIR%\cmicro`.

```
cmicro
   +-- include
   +-- kernel
   +-- mcod
   +-- template
         +-- commlink
   +-- tester
```

Furthermore all files, which are used when targeting Cmicro, are definitely stored in this directory tree. Otherwise the files are generated by the Cmicro SDL to C Compiler or by the Targeting Expert.

Prerequisites

Before starting the targeting with Cmicro it is necessary to have an SDL system designed and already tested with the help of the Simulator.

This means targeting begins when the first testing phase is finished.

All steps of targeting will be discussed in the following sub-sections.

Different Steps in the Work Flow

All the different operation steps listed here will completely supported by the Targeting Expert (see chapter 60, *The Targeting Expert*).

1. To start the Targeting Expert, select the entry *Targeting Expert* in the Organizer's *Generate* menu.
2. Select pre-defined integration settings or <*user defined*> for a new integration. See "Pre-defined Integration Settings" on page 2902 in chapter 60, *The Targeting Expert*.
   - select the Code Generator Cmicro (if not automatically done)
   - select the compiler (if not automatically done)

   The Cmicro SDL to C Compiler will automatically be invoked and generates the following files (assumed that separation is not selected):

sdl_cfg.h	the automatic configuration
<systemname>.c	the SDL system
<systemname>.ifc	the environment header file
<systemname>_gen.m	the sub-makefile (Targeting Expert)
<systemname>.xrf	the X-References (not used)
<systemname>.sym	the symbol file (SDL Target Tester)

3. Specify the compiler, linker and make settings (if not automatically done).

   Please view "Configure Compiler, Linker and Make" on page 2856 in chapter 60, *The Targeting Expert*.
4. Configure and scale the generated C code and the Cmicro Library.
   - For information about the compilation flags and their interdependencies please view "Compilation Flags" on page 3394.
   - For more details how to use the Targeting Expert please view "Configure and Scale the Target Library" on page 2872 in chapter 60, *The Targeting Expert*.
5. Copy template files.

Several files of the Cmicro Package are delivered as template files. This is done because lots of things can only be done by the user as they must fit to the user's SDL system. The template files give easy to use C functions including help to adapt them to the user's needs.

All the files `mk_stim.c`, `mk_user.c` and `mk_cpu.c` should be copied into the project's directory tree (generated by the Targeting Expert). All these files are stored in the `template` directory.

If the SDL Target Tester should also be used it is necessary to copy the files `mg_dl.c` and the files describing the preferred communications link (e.g. the file `8051_v24.[ch]` if an 8051 micro controller and a V.24 interface should be used) into the project directory, too. The file `mg_dl.c` is stored in the `template` directory and the communications link files are stored in `template/commlink` if contained in the Cmicro Package delivery. Please view the sub-section "The Communications Link's Target Site" on page 3618 to get information about how to create an own communication link.

Please view also "Source Files" on page 2861 in chapter 60, *The Targeting Expert* to get information on how to add more files to the list of files to be compiled. The Targeting Expert will automatically add these files to the makefile.

6. Full make the complete system.

   It is probably necessary to do an environment connection first (see "Connecting the SDL Environment" on page 3391).

7. Download and execute the executable.

   The following sections in the Targeting Expert's manual (chapter 60, *The Targeting Expert*) should be checked to find out how these steps can be eased:

   – "Download Application" on page 2880

   – "Test Application" on page 2880 (which is probably the target executable itself)

Connecting the SDL Environment

From the Cmicro Package's point of view the SDL environment is represented in template C functions. A short overview of this C functions

is given here. For further information please view "Bare Integration" on page 3437.

- xInitEnv():
  This function is given as a template in the file `env.c`. It needs to be filled any time it is necessary to initialize specific hardware components on the target.

- xCloseEnv():
  `xCloseEnv()` is given as a template in `env.c`, too. It is only necessary to fill this function if the SDL system can perform a system stop.

- xInEnv():
  `xInEnv()` is basically generated by the Targeting Expert into the file `env.c`. This means that all the signals coming from the environment are already inserted into the C code of this function.

Note:

All the modifications of `env.c` should only be done between the comments `/* BEGIN User Code */` and `/* END User Code */`. All the other modifications will be lost when re-generating the `env.c` file.

- xOutEnv():
  For all the signals to the environment the Targeting Expert generates the C function `xOutEnv()` into the file `env.c`. All the signals to the environment are included into the C code.

Note:

All the modifications of `env.c` should only be done between the comments `/* BEGIN User Code */` and `/* END User Code */`. All the other modifications will be lost when re-generating the `env.c` file.

- User Defined Actions for System Errors – the ErrorHandler:
  The Cmicro Package supports the handling of detected errors by calling the C function `ErrorHandler()` in the module `mk_user.c`. The user needs to redirect the error messages to his target hardware. I.e. display the errors on the hardware.

The compilation and linkage of the environment functions can be prevented by using the flags given in "Compilation Flags" on page 3394.

Different Forms of Target Integration

Different forms of target integration, that is, integrating generated C code, are distinguished:

- Bare Integration: There is no operating system available on the target machine. The `main()` function of Cmicro can be used.
- Light Integration: An operating system is used and the complete SDL system executes in one operating system task. The SDL task communicates with the environment by using the communication resources of the operating system.
- System partitioning: An operating system is used, the SDL system may execute in different CPUs, or the processes of the SDL system execute in different OS tasks.
- Tight Integration: An operating system is used, the SDL system executes in one CPU, and the processes of the SDL system execute in different tasks.

Bare and Light integration represent the most easiest form of integration.

Bare integration is described in "Bare Integration" on page 3437.

Light integration is described in "Light Integration" on page 3466.

Tight integration can be performed with Cmicro, but the complexity makes it difficult to describe the integration in this section. It is a generic solution available, which guides thought the integration.

Please contact Telelogic local sales office/Professional Services for further information.

Compilation Flags

Compilation flags are used to decide the properties of the Cmicro Library and the generated C code. Both in the Cmicro Library and in the generated code `#ifdef`'s are used to include or exclude parts of the code.

The switches used can be grouped:

1. Flags defining the compiler
2. Flags defining the properties of a compiler
3. Flags defining the properties of the Cmicro Library
4. Flags defining the implementation of a property

The first two groups are discussed in "Adaptation to Compilers" on page 3430.

The remaining two groups are discussed under "Manual Scaling" on page 3394 and "Automatic Scaling Included in Cmicro" on page 3426.

Flag naming conventions:

- `XMK_ADD_`: include optional parts
- `XMK_USED_`: automatically include parts of the Cmicro SDL to C Compiler. Note: Do not switch off by hand.
- `XMK_USE_`: manual scalings
- `XMK_MAX_`: manual scalings

Use the default settings of these flags in order to reduce potential problems.

Manual Scaling

Users are able to scale some features of the Cmicro Library and the Cmicro Kernel in order to optimize the generated code. All the flags discussed in this section are used throughout the complete SDL system, therefore it is not possible to define flags for processes separately.

> **Note:**
> The whole SDL system must always be generated.

The manual scalings have to be done in the file `ml_mcf.h`. They are divided into the following groups:

1. Cmicro Kernel/Library
   - Kernel
   - Signals
   - Timers
   - Error checks
2. Support of SDL Constructs
   - Predefined sorts
   - Size of variables
   - Use of memory
3. SDL Environment
4. SDL Coder
5. SDL Target Tester
   - Initialization
   - Trace
   - Record and Play
6. Communication Link and Compiler

In the following subsections, the way to configure the Cmicro Package by hand and all the available flags are described.

The user is asked to use the Targeting Expert to configure the Cmicro Package. A description of how to use the targeting Expert can be found in "Targeting Work Flow" on page 2852 in chapter 60, *The Targeting Expert*.

To configure the Cmicro Package by hand, the user has to modify the file `ml_mcf.h`.

To use a Cmicro Package feature just define it in `ml_mcf.h` like

```
#define FLAG_NAME_AS_DESCRIBED_BELOW
```

Sometimes a flag just carries a value. In this case the user has to modify the value in `ml_mcf.h`

```
#define FLAG_CARRYING_VALUE  VALUE
```

Cmicro Kernel/Library

Kernel

General

- XMK_USE_KERNEL_INIT

  When this flag is defined, the Cmicro Kernel `memsets` all variables of the SDL processes to 0 before the process is created. This is useful to spare some ROM memory but has the disadvantage of a longer process creation phase. The ROM used by the `yDef_SDL_*` functions can be spared in this case where the initialization with 0 is appropriate.

Initial setting	set

- XMK_USE_SDL_SYSTEM_STOP

  Normally, it is unnecessary to have an SDL system stop in micro controller applications. An SDL system stop occurs if no living process in the SDL system exists and all queues are empty. This occurs if all process instances have executed an SDL stop and no create process action is open (create signal in any queue).
  Defining `XMK_USE_SDL_SYSTEM_STOP` means that the C function `main` returns correctly with exit(...). To avoid this overhead in the Cmicro Kernel, the user should not define the above flag.

Default setting	unset

- XMK_SYSTEM_INFO

  If this flag is set, some functions available to get information about the SDL system during runtime. See "Functions to get System Information" on page 3500

Initial setting	unset

- XMK_USE_NO_AUTO_SCALING

  When this flag is defined, automatic scaling of SDL features is prevented. An example of flag usage is implementing a change to the SDL system which does not require re-compilation of the Cmicro Library. Note that if the number of processes, signals or timers is changed, the Cmicro Library must be re-compiled. The definitions contained in `sdl_cfg.h` are overwritten in `ml_typ.h`.

Initial setting	unset

- XMK_USE_KERNEL_WDTRIGGER

  If this flag is defined, the Cmicro Kernel calls the C function `WatchdogTrigger` before executing a transition. This function is present as a template in the Cmicro Library source files.

Initial setting	unset

Kernel Limits

- XMK_USE_HUGE_TRANSITIONTABLES

  Per default the Cmicro Package can handle up to 252 different transitions per process. If the SDL system exceeds this size, it is necessary to set this flag. See "Transition Table" on page 3332 for further information.

Initial setting	unset

- XMK_USE_MORE_THAN_250_SIGNALS

  Per default the Cmicro Package is designed to handle up to 250 user defined signals. When using larger SDL systems it is necessary to define this flag. This results in larger RAM and ROM occupation and should only be done if necessary (see `xmk_T_SIGNAL`).

Initial setting	unset

- XMK_USE_MORE_THAN_256_INSTANCES

  Per default the Cmicro Package handle up to 256 instances per process type. If the system uses more than 256 instances per type it is necessary to set this flag.

Initial setting	unset

- XMK_USE_PAR_GREATER_THAN_250

  Per default the Cmicro Package is able to handle signal parameters up to a length of 250 octets. If greater signal parameters are used (at least once) this flag needs to be set. Please view `xmk_T_MESS_LENGTH` in `ml_typ.h` for further information.

Initial setting	unset

Signal Structure

- XMK_USE_SENDER_PID_IN_SIGNAL

  When this flag is defined a sender pid is included in each signal instance. It is possible to omit the sender pid if the system contains no to SENDER or to PARENT addressing.

Initial setting	set

- XMK_USE_RECEIVER_PID_IN_SIGNAL

  When this flag is defined a receiver pid is included in each signal instance. It is possible to omit the receiver pid if the user writes a C function `xRouteSignal()` which is given as a template in `mk_user.c`. Each signal type is then mapped to a unique receiver. It is recommended to define this flag in small systems where unique receivers exist for each signal type. It is important to note that in the case of dynamically created processes an internal create signal is used. If there are any (x, N) where N > 1, declarations in the system, the create signal cannot contain the receiver pid. The receiver pid is necessary for correct creation of processes. Never leave it out when using dynamic process creation.

Initial setting	set

- XMK_MSG_BORDER_LEN

  The value of this macro gives the length of signal parameters carried inside the signal. Note: If the parameter length exceeds this value a memory allocation is performed and the signal parameters are copied into this buffer. The pointer to the allocated buffer is carried inside the signal's structure (see `ml_typ.h` for the typedef `xmk_T_SIGNAL`).

Initial value	4

Signal Handling

- XMK_USE_xmk_SendSimple

  This flag enables the output C function `xmk_SendSimple()` defined in the Cmicro Kernel. If the SDL system contains several signals without parameters and priorities, it is useful to set this flag in order to select the more optimal output C function, `xmk_SendSimple()`, in the Cmicro Kernel.

Initial setting	set
Reset	XMK_USE_SIGNAL_PRIORITIES

- XMK_USE_SAFE_ADDRESSING

  Setting this flag includes a special entry in the signal queue which ensures the handling of sender and offspring in the start transition.

Initial setting	unset

Signal Queue

- XMK_USE_STATIC_QUEUE_ONLY

  Create signals only from the static memory. The static memory is predefined with XMK_MAX_SIGNALS.

Initial setting	set

Reset	XMK_USE_STATIC_AND_DYNAMIC_QUEUE

- XMK_USE_STATIC_AND_DYNAMIC_QUEUE

  Create signals from the static memory which is predefined with XMK_MAX_SIGNALS. If more signals have to be inserted memory is allocated from the dynamic memory pool.

Initial setting	unset
Reset	XMK_USE_STATIC_QUEUE_ONLY

- XMK_MAX_SIGNALS

  In the Cmicro Package, the SDL queue is physically implemented as one queue for all processes. The define discussed in this section represents the maximum amount of signals in the static signal instance memory pool (see "Dynamic Memory Allocation" on page 3450 for more information). It may be difficult to evaluate the maximum entries required during run-time because this totally depends on how the SDL system is specified, and target hardware performance. It is for example, impossible to state how much time hardware requires to process an SDL signal.
  By examining the SDL system it can be determined which processes have a long transition time and which processes send or receive more than one signal. Estimate by trying out worst case situations.
  A first estimation is also possible by calculating:
  maximum amount of process instances * 3
  For a more exact estimation the user should use the profiler contained in the SDL Target Tester to obtain the necessary information on how many entries are used during run time.
  Another method of helping to determine the maximum amount of signals required by the system is to use the exception handling mechanism offered by the `ErrorHandler`, i.e. when the queue is full and another signal is to be inserted then the `ErrorHandler` function is called.

Initial value	20

Note:

Using dynamic memory allocation does not prevent the user from estimating the required memory for the queue. Sooner or later problems arise (e.g. memory fragmentation) if dynamic memory management is used frequently. For this reason the Cmicro Package avoids where possible the use of dynamic memory management.

Light Integration

- XLI_LIGHT_INTEGRATION

  If this flag is set, some functionality is included to make a light integration easier.

Initial setting	unset
Automatic set	XMK_USE_INTERNAL_QUEUE_HANDLING XLI_LIGHT_INTEGRATION

- XMK_USE_INTERNAL_QUEUE_HANDLING

  This flag make additional functions for queues the handling available which are needed for light integrations.

Initial setting	unset
Reset	XLI_LIGHT_INTEGRATION

- XLI_INCLUDE

  The value of this is the name of the header file which is included in ml_typ.h when XLI_LIGHT_INTEGRATION is set. This file contains the definitions of the light integration macros. See "Define the macros which are needed in the task function." on page 3468

Initial setting	"li_os_def.h"

Tight Integration

The complexity of a tight integration is very high. But there is a generic solution available, which guides you through the integration.

Please contact Telelogic local sales office/Professional Services for further information about this solution.

- XTI_USE_INTERNAL_OUTPUT

  For more information about this contact local Professional Services

Initial setting	unset

- XTI_USE_LOCK_IN_CREATE

  For more information about this contact local Professional Services

Initial setting	unset

Error Handler

- XMK_USE_MAX_ERR_CHECK

  When this flag is defined, additional error checks are included in the generated code of the Cmicro Library. For further details, see the appropriate section on “errors and warnings”. For example the Cmicro Kernel calls the ErrorHandler if a signal is sent to an undefined process (i.e. undefined pid value).

Initial setting	set
Reset	XMK_USE_MIN_ERR_CHECK XMK_USE_NO_ERR_CHECK

- XMK_USE_MIN_ERR_CHECK

  In comparison to `XMK_USE_MAX_ERR_CHECK` this flag only includes a basic set of error checks.

Initial setting	unset
Reset	XMK_USE_MAX_ERR_CHECK XMK_USE_NO_ERR_CHECK

- XMK_USE_NO_ERR_CHECK

  When this flag is defined, all error checks are excluded. It is recommended not to use this flag until the post testing phase of the implementation, where it can be reasonably assumed that no errors remain. For further details see the appropriate section on “errors and warnings”.

Initial setting	unset
Reset by	XMK_USE_MAX_ERR_CHECK XMK_USE_MIN_ERR_CHECK

- XMK_USE_MON

  If the target executable runs on an environment with monitor functions, the module `ml_mon.c` can be included to have access to functions like `xxmonhexasc()` etc.

Initial setting	unset

- XMK_ADD_PRINTF

  When this flag is defined some additional `printf` C function calls are compiled giving users more information about the internal work of the system. The `printf` function can be switched on separately for Cmicro Kernel, SDL Target Tester and SDL application. At a lower level, it can be switched on separately for each C module. Look at the defines in `ml_typ.h`, which are all named as `XMK_ADD_PRINTF_*`.
  This flag must be undefined when compiling for the target, except in the case where there is a `stdio` implemented on the target. For correct compilation, the user must also set XMK_ADD_STDIO. If the user wishes to implement user specific `printf` functionality then this flag need not be set.

Initial setting	unset
Automatic set	XMK_ADD_STDIO

Reactions on Warnings

- XMK_WARN_ACTION_HANG_UP

If this flag is defined, the default behavior, if a warning is detected during SDL execution, is defined as “program hang up”.

The user might choose this reaction when there is no output device (like `printf`) available in the SDL program environment.

The user should notice that a warning can lead to an illegal system behavior. As an example, this might come true for an implicit signal consumption. The system then hangs but the user might perhaps not be able to see the reason why this occurs. As a result, it is recommended to trace for warnings also.

Initial setting	set
Reset	XMK_WARN_ACTION_PRINTF XMK_WARN_ACTION_USER

- XMK_WARN_ACTION_PRINTF

  By defining this flag, the user chooses that in the case of a warning, a `printf` function call with an appropriate error text should occur. Please see `XMK_WARN_ACTION_HANGUP` also.

Initial setting	unset
Reset	XMK_WARN_ACTION_HANG_UP XMK_WARN_ACTION_USER

- XMK_WARN_ACTION_USER

  By defining this flag, the user chooses that in the case of a warning, a user defined function should be called. The user’s function name must then be specified with `XMK_WARN_USER_FUNCTION`. Please see `XMK_WARN_ACTION_HANGUP` also.

Initial setting	unset
Reset	XMK_WARN_ACTION_HANG_UP XMK_WARN_ACTION_PRINTF

- XMK_WARN_USER_FUNCTION

If the flag XMK_WARN_ACTION_USER is defined, then the user must define the name of a C function with this macro. Please see XMK_WARN_ACTION_HANGUP also.

Initial setting	user_function()

Reaction on Errors

- XMK_ERR_ACTION_HANG_UP

  If this flag is defined, the default behavior, if a fatal system error is detected during SDL execution, is defined as “program hang up”.

  The user might choose this reaction when there is no output device (like printf) available in the SDL program environment.

  The user should notice that if a fatal system error is ignored, this usually leads to an illegal system behavior. As an example, this might come true for any use of null pointer values, for which there is an error check. It is strongly recommended to trace for system errors and warnings.

Initial setting	set
Reset	XMK_ERR_ACTION_PRINTF XMK_ERR_ACTION_USER

- XMK_ERR_ACTION_PRINTF

  By defining this flag, the user chooses that in the case of a system error, a printf function call with an appropriate error text should occur. Please see XMK_ERR_ACTION_HANGUP also.

Initial setting	unset
Reset	XMK_ERR_ACTION_HANG_UP XMK_ERR_ACTION_USER

- XMK_ERR_ACTION_USER

  By defining this flag, the user chooses that in the case of an system error, a user defined function should be called. The user’s function name must then be specified with XMK_ERR_USER_FUNCTION. Please see XMK_ERR_ACTION_HANGUP also.

Initial setting	unset
Reset	XMK_ERR_ACTION_HANG_UP XMK_ERR_ACTION_PRINTF

- XMK_ERR_USER_FUNCTION

  If the flag `XMK_ERR_ACTION_USER` is defined, then the user must define the name of a C function with this macro. Please see `XMK_ERR_ACTION_HANGUP` also.

Initial setting	user_function()

Timer Scaling

If there are timers used within the SDL system, the timers have to be scaled.

- XMK_USE_TIMER_SCALE

  The timer units can be scaled within the Cmicro Kernel. This means that the factor given with XMK_USE_TIMER_SCALE_FACTOR is used.

Initial setting	unset

- XMK_USE_TIMER_SCALE_FACTOR

  The factor modifies the expiration of timers. Is a value of 1 given here, it means that the expiration time is un-modified. A value of 100 increase the expiration time with a factor of 100 with other words the timer needs 100 times longer.

Initial value	100

- XMK_TIMERPRIO

  An expired timer is handled like a signal inside the Cmicro Kernel and inserted into the signal queue. In this way there has to be a priority level if signal priorities are used. The XMK_TIMERPRIO gives the priority for all expired timers. This default value can lay between 0 and 250.

Initial value	100

Timer queue

- XMK_USE_GENERATED_AMOUNT_TIMER

  The code generator evaluates the amount of timers that are declared in the system. The result of this evaluation is then, after code generation, defined with `XMK_MAX_TIMER_INST` in the file `sdl_cfg.h`. Timer instances are, usually, implemented as a C array with Cmicro. If the `XMK_USE_GENERATED_AMOUNT_TIMER` flag is set, then the timer array is dimensioned with the evaluated amount (`XMK_MAX_TIMER_INST`).

  When timers with parameters are in the system, the automatically generated value `XMK_MAX_TIMER_INST` can be used to pre-define one timer instance of a timer declaration. If there are then more timers to be instantiated, dynamic memory allocation must take place. This cannot be evaluated by the code generator.

  As a result, if there are timers with parameters in the system, the user should think about how many timer instances there could be during execution and should decide upon the maximum amount by himself. In this way, memory consumption and performance can be balanced.

Initial value	set

- XMK_MAX_TIMER_USER

  This macro is used for internal purposes. The meaning of it is the opposite of `XMK_USE_GENERATED_AMOUNT_TIMER`.

  The flag is set when `XMK_USE_GENERATED_AMOUNT_TIMER` is not set. The flag `XMK_MAX_TIMER_USER` is unset if `XMK_USE_GENERATED_AMOUNT_TIMER` is set.

Initial value	unset

- XMK_MAX_TIMER_USER_VALUE

  If `XMK_MAX_TIMER_USER` is set, the value `XMK_MAX_TIMER_USER_VALUE` becomes meaningful. With this val-

ue, the user may specify how many timer instances should be predefined during compilation time. The predefined timer instances offer the advantage that no dynamic memory allocation is to be done for them. If the predefined amount of timer instances is exceeded, then dynamic memory allocation will occur. As a result, it is up to the user how he balances static and dynamic memory for timer instances by changing this value.

Initial value	20

Execution Time

- XMK_USE_CHECK_TRANS_TIME

  When this flag is defined the time duration for each executed transition is checked against a predefined duration. If the executed duration is longer than the predefined duration set by the `XMK_TRANS_TIME` flag, the `ErrorHandler()` is called. The flag is only available when using a non preemptive Cmicro Kernel. No additional hardware is necessary as the evaluation is based on the SDL time value `NOW` which is provided by the same hardware source as for the system clock.

Initial setting	unset
Automatic set	XMK_TRANS_TIME

- XMK_TRANS_TIME

  This macro specifies the time duration that a transition execution time is compared with. The value must be in the target system time units, e.g. if the target's system step is 0,002 sec, the value 100 specifies a duration of 0.2 sec.

Initial value	20
Depend on	XMK_USE_CHECK_TRANS_TIME

Signal Priorities

In the standard configuration of the Cmicro Kernel no signal priorities are used. So each signal sent is inserted at the end of the signal queue.

- XMK_USE_SIGNAL_PRIORITIES

When this flag is defined, the header of each signal which is sent via the SDL queue has an additional entry, namely “priority”. This is used to modify the ordering of signals in the queue.
This flag should be used in combination with using the #PRIO directive. It works for all the different scheduling methods.
If signal priorities are enabled, signal handling `xmk_SendSimple` has to be switched of.

Initial setting	unset
Automatic set	xDefaultPrioSignal XMK_CREATE_PRIO
Reset	XMK_USE_xmk_SendSimple

- xDefaultPrioSignal

  If a signal has not got a priority assignment within the SDL system, it is handled with the default value `xDefaultPrioSignal`. A value between 0 (highest priority!) and 250 should be entered.

Initial value	100
Depend on	XMK_USE_SIGNAL_PRIORITIES

- XMK_CREATE_PRIO

  Create signals can not be assigned a signal priority within the SDL system. As these signals are also handled over the signal queue, a default priority is necessary when signal priorities are selected (i.e. `XMK_USE_SIGNAL_PRIORITIES` is defined). A value between 0 (highest priority!) and 250 should be entered.

Initial value	1
Depend on	XMK_USE_SIGNAL_PRIORITIES

Preemption

Normally, the Cmicro Kernel is configured so that the simple scheduling policy is used, i.e. transitions of SDL processes are non interruptible.

Note:

The Cmicro preemptive kernel is only available if an according license is available.

- XMK_USE_PREEMPTIVE

  Usually the Cmicro Kernel is configured to “non preemptive”. This means, that each transition must execute to its end, before the next transition can be processed.
  This means, that the nextstate symbol of a process is executed, before the next SDL input can be handled. On the other hand, “preemptive scheduling” is useful, if an SDL process must directly execute when an external event from an interrupt source is detected and sent to the SDL system. In this case, an executing transition is possibly suspended, and another transition can be executed.
  The kernel makes decisions based on the defined priority of processes when to schedule to another process. If the receiver of a signal which has been received from the environment (or a process inside the SDL system) has a higher priority than the currently executing process, the new signal is treated immediately.

Initial setting	unset
Automatic set	MAX_PRIO_LEVELS xDefaultPrioProcess XMK_USE_RECEIVER_PID_IN_SIGNAL
Reset	XMK_USE_CHECK_TRANS_TIME

- MAX_PRIO_LEVELS

  This defines the amount of process priority levels in the SDL system. It is of relevance only, if preemption is selected and must be equal to the lowest process priority plus 1. The lowest priority has the highest value.

Initial value	1
Depend on	XMK_USE_PREEMPTIVE

Caution!

A run-time error occurs if this definition is wrong. If the error checks are enabled, a check is made by the Cmicro Kernel and the C function `ErrorHandler()` is called if the check fails.

- xDefaultPrioProcess

  If there is an SDL process with no process priority assigned to it, this process will be handled with the `xDefaultPrioProcess`. It is mandatory that the `xDefaultPrioProcess` is in the range of 0 to (`MAX_PRIO_LEVELS` -1).

Initial value	0
Depend on	XMK_USE_PREEMPTIVE

SDL Target Tester

Initialization

- XMK_ADD_MICRO_TESTER

  This is the main flag to enable or disable the SDL Target Tester. Use this flag if the addition of the SDL Target Tester features is wanted. The following features are selected by this flag:

  – Trace into a file.
  – Trace into a buffer on target side.
  – Trace via a serial interface.
  – the Cmicro Recorder

Initial setting	unset
Automatic set	XMK_USE_MAX_ERR_CHECK

Reset when unset	XMK_ADD_MICRO_COMMAND, XMK_USE_DEBUGGING, XMK_ADD_PROFILE, XMK_ADD_MICRO_ENVIRONMENT, XMK_WAIT_ON_HOST, XMK_ADD_SDLE_TRACE, XMK_ADD_SIGNAL_FILTER, XMK_USE_COMMLINK, XMK_ADD_MICRO_TRACER, XMK_ADD_TEST_OPTIONS, XMK_USE_SIGNAL_TIME_STAMP, XMK_ADD_MICRO_RECORDER

- XMK_WAIT_ON_HOST

  During the start up of the target executable (see `xmk_MicroTesterInit()` in the module `mk_main.c`), the target can wait for the host's run-time configuration. If this is not needed, the `XMK_WAIT_ON_HOST` flag has to be undefined.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_ADD_PROFILE

  This flag is to be defined if the profiler is to be used. With the profiler, it is possible:

  – to trace the SDL queue concerning it's traffic load,
  – to trace the number of timers which exist in parallel (not included in this version of the Cmicro Package).
  – to trace the execution time process transitions

  The profiler can be used independently from the SDL Target Tester functions. By using a debugger the following values may be inspected:

  `int xmk_max_q_cnt,`

  representing queue-dimensioning, and

  `int xmk_act_q_cnt,`

  representing the maximum traffic load of the queue at any time during the execution.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_USE_DEBUGGING

  If the debugging functions should be included to the target executable, this flag needs to be defined.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_ADD_MICRO_COMMAND

  This flag adds the command interface of the SDL Target Tester (please view chapter 68, *The SDL Target Tester*).

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_ADD_MICRO_ENVIRONMENT

  To allow the signal exchange with the external environment connected to the SDL Target Tester, this flag needs to be defined.

  The external environment can be a GUI application or a cmdtool for example.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

Trace

Trace Scaling

- XMK_ADD_MICRO_TRACER

  This flag enables the trace of the SDL execution and trace of system events to any device.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER
Reset when unset	XMK_USE_SIGNAL_TIME_STAMP

- XMK_ADD_SDLE_TRACE

  This flag enables the trace of SDL symbols in the SDL Editor.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_USE_COMMLINK

  If this flag is defined the trace via the specified communications link is enabled. The trace is handled with the SDL Target Tester's data link layer. See the module `mg_dl.c` for further information.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_USE_AUTO_MCOD

  Is this flag defined, the host get a further message for the message coder. Depending on this flag the Targeting Expert insert the entry 'USE_AUTO_MCOD yes' into the sdtmt.opt file. The entrys LENGTH_, ENDIAN_, and ALIGN_ are ignored. For more information view chapter 68, *The SDL Target Tester*.

Initial setting	set

- XMK_ADD_SIGNAL_FILTER

  This flag conditionally compiles the signal filtering mechanism of the SDL Target Tester (please view chapter 68, *The SDL Target Tester*). With the signal filter, it is possible to specify that some signals should be traced while others are not.

Initial setting	unset

Depend on	XMK_ADD_MICRO_TESTER

- XMK_ADD_TEST_OPTIONS

  By defining this flag the symbol trace of the target can be configured. See function `main()` in module `mk_user.c`.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_USE_SIGNAL_TIME_STAMP

  Defining this flag means that in the header of each signal which is sent via the SDL queue, there is an additional entry “time-stamp”. When using the standard Cmicro Library, the Cmicro Kernel sets the time-stamp to `NOW`, if an SDL signal is inserted into the queue.

  This time-stamp is especially used by the Cmicro Recorder (XMK_ADD_MICRO_RECORDER) in order to enable the replay of an SDL session in simulated real-time.

  In other cases, the user is free to modify the type of time-stamp implementation.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER
Reset when unset	XMK_ADD_REALTIME_PLAY

- XMK_MAX_PRINT_STRING

  The user can add his own trace messages to the standard trace of the SDL Target Tester. This message will be handled like a string. The maximum size of these strings are defined with `XMK_MAX_PRINT_STRING`.

Initial value	40

Buffers

- XMK_MAX_RECEIVE_ENTRIES

If the ring buffer is used this flag gives the amount of entries for the receiver buffer.

Initial value	20

- XMK_MAX_RECEIVE_ONE_ENTRY

  This flag gives the size of one entry of the receiver buffer. The message received here are SDL Target Tester commands or the Cmicro Recorder's play data. I.e. as a rule over the thumb a value of 12 + maximum signal parameter size must be given here.

Initial value	100

- XMK_MAX_SEND_ENTRIES

  When the Cmicro Tracer is used the target executable writes the trace data into a ring buffer (view `ml_buf.c`). The recommended amount of entries given here depends of the used communications link. I.e. if using a slow communications link like V.24 this amount should be increased.

Initial value	20

- XMK_MAX_SEND_ONE_ENTRY

  When the Cmicro Tracer is used the target executable writes the trace data into a ring buffer (view `ml_buf.c`). The size of one entry depends on the SDL system. As a rule over the thumb a value of 12 + maximum signal parameter size must be given here.

Initial value	100

Recorder and Play

Note:

The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

- XMK_ADD_MICRO_RECORDER

This flag adds the functions of the record mode of the Cmicro Recorder (please view chapter 68, *The SDL Target Tester*).

Initial setting	unset
Depend on	XMK_ADD_MICRO_TRACER
Reset when unset	XMK_ADD_REALTIME_PLAY

- XMK_ADD_REALTIME_PLAY

  When defining this flag, the replay of a recorded session can be performed in simulated real-time.

Initial setting	unset
Automatic set	XMK_USE_SIGNAL_TIME_STAMP
Depend on	XMK_ADD_MICRO_RECORDER

Support of SDL Constructs

Predefined Sorts

Character Strings

- XRESTUSEOFCHARSTRING

  This flag must be seen in combination with XNOUSEOFCHARSTRING, which forbids the use of SDL charstrings. If this flag is set, SDL charstrings are only supported in a restricted way. I.e. constant character buffers are used in this implementation.

  As a third possible way XRESTUSEOFCHARSTRING and XNOUSEOFCHARSTRING should both be undefined to enable a full support of SDL charstrings.

Initial setting	set
Reset by	XNOUSEOFCHARSTRING

Caution!

The Cmicro Recorder is not able to handle SDL charstrings within signal parameters.

- XNOUSEOFCHARSTRING

  The use of SDL charstrings is not possible. This can be selected for systems witch do not use SDL charstrings.

Initial setting	set
Reset by	XRESTUSEOFCHARSTRING

- XMK_MAX_SDL_CHARSTRING

  This macro gives the maximum length of charstring if XNOUSEOFCHARSTRING is defined.

Initial value	50

Predefined Sorts

- XNO_LONG_MACROS

  The setting of this flag prevents the use of predefined generators (please view "sctpredg.h, sctpred.h and sctpred.c" on page 3473). This can be done to reduce the target's memory size. But must be done for some specific compilers as these are not able to handle long macro definitions. Please view the appropriate compiler section in `ml_typ.h` ("Adaptation to Compilers" on page 3430)

Initial setting	set

- XNOUSEOFREAL

  This flag allows using real values in SDL if it is undefined.

Initial setting	set

Note:

Every multiplication of an integer value with an integer value is mapped to a multiplication of a real value with a real value. This is done because an overflow cannot be detected when using integer values.

ASN.1 Sorts

- XNOUSEOFASN1

  If this flag is undefined it is allowed to use ASN.1 data types. It is recommended not to use ASN.1 data types if possible, because the functions that are necessary to handle these types occupy much memory.

Initial setting	set
Depend on	XNOUSEOFCHARSTRING

- XNOUSEOFOCTETBITSTRING

  If this flag is undefined it is allowed the use of ASN.1 data type octetbitstring. It is recommended not to use ASN.1 data types if possible, because the functions that are necessary to handle these types occupy much memory.

Initial setting	set
Depend on	XNOUSEOFASN1

- XNOUSEOFOBJECTIDENTIFIER

  If this flag is undefined it is allowed the use of ASN.1 data type objectidentifier. It is recommended not to use ASN.1 data types if possible, because the functions that are necessary to handle these types occupy much memory.

Initial setting	set
Depend on	XNOUSEOFASN1

Error Checks

- XEREALDIV

  If this flag is defined, then a C function is used to perform division of real values, otherwise a C macro is used. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

- XEINTDIV

  If this flag is defined, then a C function is used to perform division of integer and octet values, otherwise a C macro is used. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

- XEINDEX

  If this flag is defined, then additional error checks are introduced that check the index of array at each position an array element is accessed. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

- XEFIXOF

  If this flag is defined, then additional error checks are introduced that check the conversion from real to integer. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

- XECSOP

Setting this flag includes an error check for predefined ADT operators, as the checks occupy a lot of memory.

Note:

This flag should be used for testing purposes but uses lots of code size within the target.

Initial setting	set

- XERANGE
  XTESTF

  These flags define that an SDL syntype is to be checked against its defined range. If the flags are not set, the code to perform this check is left out. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

Size of Variables

- X_COMPACT_BOOL

  This flag defines that an SDL boolean is translated to unsigned char, when it is set. Otherwise each SDL boolean is translated to integer. Usually this flag should be left set, as it is in the default setting. An exception might become true when a C compiler can handle integer values more efficient than char values.

Initial setting	set

- X_SHORT_REAL

  When this flag is defined, then each SDL real value is translated to a float in C. If it is not defined, Cmicro assumes SDL real values are C double values. The user must take care that at no place in the SDL system there is a real value used that exceeds the maximum range of the appropriate C type.

Initial setting	set

- X_LONG_INT

  By defining this switch, the SDL predefined sorts, integer and natural are translated to long, otherwise these are translated to integer. The user must take care that at no place in the SDL system there is a real value used that exceeds the maximum range of the appropriate C type.

Initial setting	unset

Use of Memory

Memory Management

- XMK_USE_SDL_MEM

  If this flag is defined, then the Cmicro Kernel can use the dynamic memory management functions contained in the `ml_mem` module (`xmk_Malloc()`, `xmk_Calloc()` and `xmk_Free()`). This is necessary if the compiler in use has no `malloc`/`free` or if the user wishes to modify the standard behavior of these functions, i.e. using a best fit, instead of first fit searching algorithm. The user should refer to "Dynamic Memory Allocation Functions – Cmicro" on page 3451 also.

Initial setting	unset
Reset when unset	XMK_USE_memshrink

- XMK_USE_MIN_BLKSIZE

  By setting this define it is possible to organize the dynamic memory allocation in a similar way as arrays in C. If all the allocated blocks in the memory occupy the same space then no fragmentation problems occur.

Initial setting	unset

- XMK_USE_memshrink

  If this flag is defined, then the `xmk_Memshrink()` function of the `ml_mem` module can be used. This function delivers the opportunity

to free the unused space of a memory block that was requested previously with `xmk_Malloc()` or `xmk_Calloc()`.

Initial setting	unset
Depend on	XMK_USE_SDL_MEM

- XMK_USE_memset

  If this flag is defined, then the `memset()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_USE_memcpy

  If this flag is defined, then the `memcpy()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_CPU_WORD_SIZE

  For allocating memory the pointer to the end of the allocated memory must be dividable by XMK_CPU_WORD_SIZE. This value is set to 8 as default, but should be decreased to 4, 2 or even 1 if the CPU's layout allows it.

Initial value	8
Depend on	XMK_USE_SDL_MEM

- XMK_MEM_MIN_BLKSIZE

  The requested block size is rounded to a value of XMK_MEM_MIN_BLKSIZE, if the XMK_MEM_MIN_BLKSIZE is greater than the requested block size.

Initial value	64
Depend on	XMK_USE_MIN_BLKSIZE

- XMK_MAX_MALLOC_SIZE

  If the Cmicro memory functions (`xmk_Malloc()`, `xmk_Calloc()` and `xmk_Free()`) are used, the buffer that is used for dynamic memory allocation is to be initialized with `xmk_MemInit()`. The size of the buffer can be given here.

Initial value	1024
Depend on	XMK_USE_SDL_MEM

String Functions

- XMK_USE_strcpy

  If this flag is defined, then the `strcpy()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_USE_strncpy

  If this flag is defined, then the `strncpy()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_USE_strcmp

  If this flag is defined, then the `strcmp()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_USE_strlen

  If this flag is defined, then the `strlen()` function of the `ml_mem` module is compiled.

Initial setting	unset

SDL environment

- XMK_USE_xInitEnv

  This flag enables the C function xInitEnv() which is generated as a template in file env.c by the Targeting Expert.

Initial setting	set

- XMK_USE_xInEnv

  This flag enables the C function xInEnv() which is generated as a template in file env.c by the Targeting Expert.

Initial setting	set

- XMK_USE_xOutEnv

  This flag enables the C function xOutEnv() which is generated as a template in file env.c by the Targeting Expert.

Initial setting	set

- XMK_USE_xCloseEnv

  This flag enables the C function which is generated as a template in file env.c by the Targeting Expert.

Initial setting	set

- XMK_USE_SEND_ENV_FUNCTION

  This flag make it possible to use an alternative function to send signals to the environment. This is needed if the used compiler does not support re-entrant functions. The disadvantage of this implementation exists in the missing error checks. See "Alternative Function for sending to the Environment" on page 3502

Initial setting	unset

- XMK_ADD_STDIO

  The Cmicro Package is mainly intended for target implementation. However on some occasions the user may wish to use OS features. Normally this flag is undefined when compiling for the target system but may be defined if `stdio` is available and required on the target system.
  Defining `XMK_ADD_STDIO` without setting `XMK_ADD_PRINTF` allows the user to exclude the default `printf`s of the Cmicro Library and instead implement user defined `printf`s.

Initial setting	unset
Reset when unset	XMK_ADD_PRINTF

Automatic Scaling Included in Cmicro

The flags described in this section are automatically generated into the file `sdl_cfg.h` by the Cmicro SDL to C Compiler. The purpose is to exclude parts of the C code in order to reduce the generated code. Features or functions which are not required in the SDL description produce no (or only a small) overhead in the generated code.

If the user does not wish to employ automatic scaling facilities, for example if test and debugging proves too difficult, then simply define the flags XMK_USE_NO_AUTO_SCALING.

All the flags discussed in this section are used on the whole SDL system i.e. it is not possible to define flags for processes separately.

The Targeting Expert will use these flags, too, to optimize and ease the manual scaling.

- XMK_USED_ONLY_X_1

  If this flag is generated as defined, then there is no process defined in the system, where N is > 1 in the process declaration (x, N). This allows memory to be saved as the overhead for process addressing in the system is reduced. The need for numbering of process instances is also eliminated.

- XMK_USED_DYNAMIC_CREATE

If this flag is generated as defined, the SDL system uses the dynamically created process feature of SDL. (It only requires one create-symbol in the SDL system to enable this flag to be set). This of course constitutes a bigger Cmicro Kernel.

- XMK_USED_DYNAMIC_STOP

  If this flag is generated as defined, the SDL system uses the dynamic process stop feature of SDL. (It only requires one stop-symbol in the SDL system to enable this flag to be set). This of course constitutes a bigger Cmicro Kernel.

- XMK_USED_SAVE

  If this flag is generated as defined, the SDL system uses the save feature of SDL. Using save means that there has to be additional overhead in each signal stored in the SDL queue (each signal is tagged as save or not save).
  The save construct, although a useful construct for manipulation of the SDL FIFO queue mechanism, unfortunately imposes a large code overhead. The user should avoid the utilization of this construct where possible.

- XMK_USED_TIMER

  If there is a minimum of one timer declaration in the system, then this flag is generated as defined. If no timer declarations exist in the SDL system, the timer handling functions are excluded.

Note:

It is possible to implement a user defined timer model by un-defining these flags and defining some other macros.

- XMK_HIGHEST_SIGNAL_NR

  This define gives the amount of signals and timers used in the SDL system in sum. It is used to scale internal buffers of the Cmicro Package.

- XMK_USED_SIGNAL_WITH_PARAMS

  If no signals with parameters are defined in the SDL system, then this flag is generated as defined which reduces the overall code size.

Note:

It is not recommended in each case (because it is not SDL conform), but is possible to send parameters via global parameters by using C code in SDL.

- XMK_USED_TIMER_WITH_PARAMS

  This define is generated if there is at least on timer with parameter used in the system. If this is the case, additional functionality for this construct is added when the kernel is compiled. The timer operations will result in producing more overhead, both symbol by symbol and overhead in the kernel compared with the case if there are no timers with parameters defined.

- XMK_USED_SENDER, XMK_USED_PARENT, XMK_USED_OFFSPRING, XMK_USED_SELF

  The above flags are generated as defined if the user uses the appropriate addressing construct within SDL, that is sender, parent, offspring or self.
  If “to” in output actions is not used, none of the above flags is generated as defined.
  The define `XMK_USE_PID_ADDRESSING` depends on the above flags.

- XMK_USED_PWOS

  If procedures without states are contained in the SDL system, this flag is defined. Procedures with states are not supported by the Cmicro Package.

Automatic Dimensioning in Cmicro

Some resources of the Cmicro Library are automatically dimensioned. These are described in the following subsections.

- MAX_SDL_PROCESS_TYPES

  The Cmicro SDL to C Compiler counts the amount of process types in the system and generates this define. This is used to dimension some tables used in the generated code, the Cmicro Kernel and the SDL Target Tester.

- MAX_SDL_TIMER_TYPES

The Cmicro SDL to C Compiler counts the amount of timer types in the system and generates this define. This is used to dimension some tables used in the generated code, the Cmicro Kernel and the SDL Target Tester.

- MAX_SDL_TIMER_INSTS

  The Cmicro SDL to C Compiler counts the amount of instances of timers in the system and generates this define. This is used to dimension some tables used in the generated code, the Cmicro Kernel and the SDL Target Tester.

Adaptation to Compilers

In this section the steps are explained that must be carried out in order to deal with a new C compiler which is not yet in the list of available compilers.

The following parts may be modified manually:

- In `mk_stim.c`, there are template functions as templates which have to be modified to adapt another hardware/compiler to the SDL system time (Now).
- In `mk_cpu.c`, there are templates for functions which represent hardware access. Those are required if the preemptive Cmicro Kernel is used.

The following parts can be added by using the Targeting Expert:

- A compiler specific header file `user_cc.h` which will automatically be included in `ml_typ.h`. Please view "Compiler Definition for Compilation" on page 2836 in chapter 60, *The Targeting Expert*.
- An entry in the Targeting Expert's configuration files. Please view "Compiler Definition for Compilation" on page 2836 in chapter 60, *The Targeting Expert*.

List of Available C Compilers in ml_typ.h

C compilers are to be selected by the user by choosing from an available list with the help of the Targeting Expert.

The Targeting Expert will generate a C define into the `ml_mcf.h` file. When compiling Cmicro sources, the right C compiler section in `ml_typ.h` is selected by using a `#ifdef <compilername>` construct.

The following `<compilername>` defines are currently defined in `ml_typ.h`:

Compilation switch	Meaning
GCC	The GNU C++ compiler for workstations
GNU80166	The GNU UNIX C compiler for Siemens 80C166 microcontrollers
TCC80166	The BSO/Tasking DOS C compiler for Siemens 80C166 microcontrollers
TCC80C196	The BSO/Tasking DOS C compiler for INTEL 80196 microcontrollers
IARC51	The Archimedes/IAR UNIX C compiler for INTEL 8051 microcontrollers
IARC6301	The Archimedes/IAR DOS C compiler for Hitachi 6301 microcontrollers
KEIL_C51	The Franklin/Keil DOS C compiler for INTEL 8051 microcontrollers
KEIL_C166	The Franklin/Keil DOS C compiler for Siemens 80166 microcontrollers
TMS320 MSP58C80	The Texas Instruments DOS C compiler for TMS 320C2x/C5x microcontrollers
IARC7700	The Archimedes/IAR DOS C compiler for Melps 7700 microcontrollers
HYPERSTONE	Hyperstone 5.07 C compiler with HyRTK real-time kernel
MCC68K	The MicroTech DOS C compiler for Motorola 68k microprocessors
MICROSOFT_C	The Microsoft C++ compiler
BORLAND_C	The Borland C++ compiler
ARM_THUMB	The Thumb compiler for ARM microcontrollers
ICC_HC12	The ICC12 5.0 Compiler for HC12 microcontrollers

If none of these compiler flags is defined during compilation the file `user_cc.h` (generated by the Targeting Expert) will be included in `ml_typ.h`.

The remaining part of the code of the Cmicro Library should be compilable without performing any modifications. If there are problems with adapting a new compiler or hardware please contact Cmicro technical support.

The user may however decide to define his own C compiler. This is explained in the following subsection.

Introducing a new C Compiler

Adding a new C Compiler to the Project

The first alternative to add a new user defined C compiler, is to add that compiler to a user's project. This can be achieved with the Targeting Expert.

- Give the C compiler a name for using it within Cmicro. The name should as a recommendation be in the notation of C macros, that is, it should be 8 to approximately 16 characters long in uppercase letters.
- Below the Targeting Expert's *Edit* menu there is a choice called "Add a new C Compiler" by which the new C compiler can be defined.
- It is also possible to remove the user defined C compiler later on by using this menu.

These changes are not stored within the Telelogic Tau installation, but within the user's project directory (which is the target directory that was chosen when the Targeting Expert was started).

Do the C Compiler Adaptations

- Create a new file with the name `user_cc.h`. If a C compiler is selected, which is not in the list of available C compilers in the Cmicro product, the `user_cc.h` is included automatically. This can easily be done by using the Targeting Expert's *Edit* menu "Edit compiler section".

- Take care that the right path names are specified when the C compiler is invoked. There usually should be something like a "`-I.`" option which must point to the path in which `user_cc.h` is stored.

Description of user_cc.h

The things that are to be defined by the user are:

- Is the compiler able to handle function prototypes, as defined in ANSI-C?

Yes	`#undef  XNOPROTO`
	`#define XPP(x)     x`
	`#define PROTO(x)   x`

No	`#define XNOPROTO`
	`#define XPP(x)`
	`#define PROTO(x)`

- Is the controller faster in accessing character values or integer values?

char	`#define xmk_OPT_INT char`
integer	`#define xmk_OPT_INT integer`

- Does the compiler support variables stored in registers?

Yes	`#undef  X_REGISTER` `#define X_REGISTER >register<`
No	`/* Nothing to do for X_REGISTER */`

`>register<` is the compiler specific command to store variables in registers.

- Default setting:
  `#define xptrint unsigned long`
  This setting should work for the very most compilers In a few cases it is necessary to define `xptrint` as `unsigned int`.

- Default setting:
  `#define xint32 long`
  This setting should work for the very most compilers. In a few cases it is necessary to define `xint32` as `int`.

- Is the compiler able to handle the C keyword `const` in the correct way?

Yes	`#define XCONST const`
No	`#undef XCONST`

In general, the compilers which are able to produce ROM-able code, can handle the keyword `const`. Compilers may generate false object code, if using `const`.

- Is it a UNIX system for which code is to be compiled for?

Yes	`#define XMK_UNIX`
No	

- Is it an MS Windows system for which code is to be compiled for?

Yes	`#define XMK_WINDOWS`
No	

- Critical paths must be enabled and disabled:

```
#undef  XMK_END_CRITICAL_PATH
#define XMK_END_CRITICAL_PATH  \
               if (xmk_InterruptsDisabled) \
               {\
                 xmk_InterruptsDisabled--;\
                 if (!xmk_InterruptsDisabled) \
                 { ENABLE; }}

#undef  XMK_BEGIN_CRITICAL_PATH
#define XMK_BEGIN_CRITICAL_PATH \
                 DISABLE;\
                 xmk_InterruptsDisabled++;
```

`ENABLE` and `DISABLE` are the compiler specific command to allow/prevent interrupts and must be filled.

- A #include of <stdio.h>, if supported by the compiler (not all the target compilers do). Write:

```
#ifdef XMK_ADD_STDIO
  #include <stdio.h>
#endif
```

- Which include header files must be added?

For example, for the IARC51:

```
#define "io51.h"
```

is used. For the GNU80166:

```
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <c166.h>
```

is used.

- The include header files containing the prototypes for dynamic memory allocation and string and memory functions must be added. This subsection should look like this for allowing the user to select either the memory functions from the C compiler or from Cmicro:

```
#ifdef XMK_USE_SDL_MEM
  #include "ml_mem.h"
#else
  #include "string.h"
#endif /* XMK_USE_SDL_MEM */
```

It must be checked if `malloc(), free(), calloc()` etc. are really prototyped in `string.h`, on other compilers it might be `stdlib.h`. Cmicro always calls `xAlloc()` and `xFree()` which are given as examples in the `mk_cpu.c` file below the template directory. For more information on dynamic memory allocation please view the subsection "Dynamic Memory Allocation" on page 3450.

Defining the SDL System Time Functions in mk_stim.c

The following functions exist in the module `mk_stim.c`:

- `void xmk_InitSystime(void)`
  Initialize the hardware registers to support the system time
- `void xmk_DeinitSystime(void)`
  Give up to use the system time. This normally cannot happen, but in some applications, where the SDL system is stopped and restarted again during run-time, it may prove useful.
- `void xmk_SetTime(xmk_T_TIME)`
  This function sets the system-time to the given value. Called by

Cmicro Kernel in the case when an overrun in the time value is detected.

- `xmk_T_TIME xmk_NOW(void)`
  This is the most important C function to handle SDL system-time. This function returns the absolute time, which is by default defined as a `long` value (`xmk_T_TIME`).

Usually, the above functions are conditionally compiled. To make the functions available in the target system, at least one timer in SDL must be declared. The functions are not included if there is no timer declared but duration, time or now is used in SDL. This will lead to compilation errors.

To make timers in SDL operable, absolute time must be implemented. This can be reached by using a hardware free running counter or by using a timer interrupt service routine, which clocks a global variable containing the absolute time.

Bare Integration

This section deals with functions that must be adapted by the user in order to connect the SDL system to the environment i.e. connection to target hardware, and the environment.

Implementation of Main Function

The user may decide to implement his own `main()` function body when it comes to target application in a bare integration. A default `main()` function is included in the template file `mk_user.c`.

The implementation of a first `main` function looks like this:

Example 574 ——————————————————————————

```
main ()
{
  xmk_InitQueue();
  xmk_InitSDL ();
  xmk_RunSDL ();
}
```

——————————————————————————————————

Of course this example does nothing in order to start the SDL Target Tester. The only possible way to get a very simple trace output is to define:

```
#define XMK_ADD_PRINTF
```

in `ml_mcf.h` with the help of the Targeting Expert. This will include calls to the C function `xmk_printf()` at several places in the generated C code and the Cmicro Library. The `xmk_printf()` function is available as an example in the file `mk_cpu.c`.

The user can use the `main()` functions delivered with the Cmicro Kernel to have full access to all Cmicro features. This delivered `main()` function is a template and can be modified to add or remove functionality.

Integrating Hardware Drivers, Functions and Interrupts

There is no extra handling for hardware drivers, hardware functions and interrupt service routines. It is necessary to implement an interface in the user's application. Hardware drivers, hardware functions and interrupt service routines are seen from the SDL system as the environment. The user has to implement the interface between the SDL system and the environment as it is described in the following subsections.

Critical Paths in the Cmicro Library

As with every real time application the Cmicro Library has to deal with critical paths.

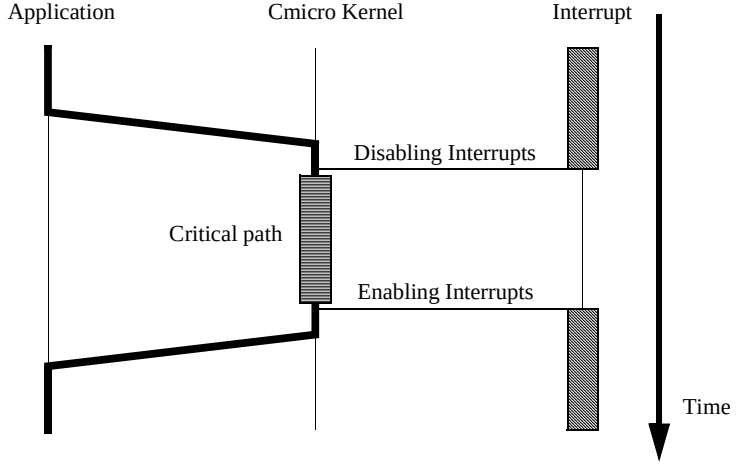

Figure 588: Critical paths

These are well defined in the source code using two macros namely `XMK_BEGIN_CRITICAL_PATH` and `XMK_END_CRITICAL_PATH` (see file `ml_typ.h`/compilersections).

The macros are expanded to compiler specific calls which disable/enables interrupts. The calls are counted using an internal variable, thus after having called `XMK_BEGIN_CRITICAL_PATH` n times `XMK_END_CRITICAL_PATH` has also to be called n times, to enable the interrupts again.

In the Cmicro Library these macros are used to disable/enable interrupts only. The application should take care not to handle interrupts without using these macros or at least evaluating the internal counter.

Caution!

When it comes to implementing the interrupt handling routines it is of the greatest importance to know about the Cmicro Library's handling of critical paths. Unpredictable results may result from a careless implementation!

Initializing the Environment / Interface to the Environment

xInitEnv()

There is one C function called `xInitEnv()` available as a template in the generated file `env.c`. The user should fill this module out appropriately, thus implementing connection to the environment (drivers, interrupt service routines and so on).

Receiving Signals from the Environment

There are two possibilities to send signals into the SDL system. In all cases, the user has to specify an SDL input symbol in the process which has to consume the signal. The possibilities are:

1. Polling external events.
   This is done by the C function `xInEnv()`, which is called by the Cmicro Kernel after each transition.

2. Directly sending signals into the SDL system.
   For example directly in an interrupt service routine. This is possible by using the `XMK_SEND_ENV()` function. This function directly operates on the SDL queue.

Caution!

For each signal sent into the system, the user must ensure that the addressed receiver process instance exists. Each pid value is checked for consistency by the Cmicro Kernel. In the case of an inconsistency the `ErrorHandler()` is called with an error message.

xInEnv()

Parameters:

```
In/Out: -no-
Return: -no-
```

In the case when no interrupt service routine is available to handle an external event, the Cmicro Kernel can poll external events. The user must use the C function `xInEnv()` in the file `env.c` generated by the Targeting Expert.

For example, a hardware register can then be checked in order to establish if its value has changed since the last call.

If an external signal is detected, then one of the `xmk_Send*()` functions is to be called thus enabling the signal to be put into the SDL queue.

Note:

There is a file called `<systemname>.ifc`, which is generated by the Cmicro SDL to C Compiler. It is necessary to include this file together with `ml_typ.h` before defining `xInEnv()`. This is necessary to have access to all the objects that are generated by the Cmicro SDL to C Compiler. The user should also make sure that this file is generated, because the analyzer's make menu contains an option for this.

Example 575: `xInEnv()` ----------

Assume, a hardware register where the value 0 is the start-value and where any other value means: data received. Two signals, namely `REGISTER_TO_HIGH`, and `REGISTER_TO_LOW` are to be defined in the SDL system, with a process receiving these. Then the user supplied C code should look like this (the `<p>` below stands for the automatically generated prefix, see the file `sdl_cfg.h`):

```
void xInEnv (void)
{
  XMK_SEND_TMP_VARS
  static state = 0;
              /* state, to detect changes in the */
              /* hardwareregister               */

 /* BEGIN User Code */
 if ((state==0) && (hw_register != 0))
 /*   END User Code */
 {
   XMK_SEND_ENV (REGISTER_TO_HIGH,
                 xDefaultPrioSignal,
                 0,
                 NULL,
 /* BEGIN User Code */
   GLOBALPID(XPTID_<p>_ReceiverName,0));
   state = 1;
 /*   END User Code */
   return;
 }

 /* BEGIN User Code */
 if ((state==1) && (hw_register == 0))
 /*   END User Code */
 {
   XMK_SEND_ENV (REGISTER_TO_LOW,
                 xDefaultPrioSignal,
                 0,
                 NULL,
/* BEGIN User Code */
   GLOBALPID(XPTID_<p>_ReceiverName,0));
   state = 0;
/*   END User Code */
    return;
  }

  return;
} /* END OF SAMPLE */
```

Caution!

The function `xInEnv()` contained in the file `env.c` is generated by the Targeting Expert. To make sure that changes done by the user will not be lost when generating again it is allowed to modify this file only between the comments `/* BEGIN User Code */` and `/* END User Code */`.

Furthermore it is not allowed to remove these comments or to add similar at other places which could especially happen when copy and paste is used during editing.

XMK_SEND_ENV()

Parameters:

```
In/Out: xPID Env_ID

        xmk_T_SIGNAL sig

        #ifdef XMK_USE_SIGNAL_PRIORITIES
          xmk_T_PRIO prio
        #endif

        #ifdef XMK_USED_SIGNAL_WITH_PARAMS
          xmk_T_MESS_LENGTH data_len,
          void xmk_RAM_ptr  p_data
        #endif

        #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
          xPID Receiver
        #endif

Return: -no-
```

It is possible to send signals directly into the SDL system by calling the C function `XMK_SEND_ENV()`, no matter whether preemption is used or not. No conflict occurs, if an interrupt service routine uses a `XMK_SEND_ENV()` function parallel to the C function `xInEnv()`.

This function is to be called when a signal is to be sent into the SDL system, e.g. within the users `xOutEnv` function. It must be called with one more parameter than the `xmk_Send` function, which is the first parameter `Env_ID`. This parameter must be set to `ENV` by the user.

The macro internally uses some variables which are to be declared before the macro can be used. For example in the xOutEnv function the `XMK_SEND_TMP_VARS` macro must be introduced for declaring these variables.

The function is implemented as a macro in C.

Use the template in the previous subsection to see details how to use the `XMK_SEND_ENV()` functions.

Sending Signals to the Environment

There are several possibilities to send signals to the environment:

1. Using simple SDL output.
   The Cmicro Kernel is involved in the output operation.
2. Using the `#EXTSIG` directive in the SDL output.
   The user can define his own output operation, like writing to a register in a single in-line-assembler command.
3. Using the `#ALT` directive in the SDL output.
   A variant of alternative 2.

Alternative 1 is the most SDL like alternative because it does not use non SDL constructs, like directives. This alternative should be selected, where possible because it makes the diagrams more SDL like and MSCs more readable.

Alternative 2 and 3 are probably the alternatives with higher performance.

Alternative 1 is described below. Alternative 2 and 3 are already well described in chapter 66, *The Cmicro SDL to C Compiler*.

xOutEnv()

Parameters:

```
   In/Out:
            xmk_T_SIGNAL        xmk_TmpSignalID

            #ifdef XMK_USE_SIGNAL_PRIORITIES
            xmk_T_PRIO          xmk_TmpPrio
            #endif

            #ifdef XMK_USED_SIGNAL_WITH_PARAMS
              xmk_T_MESS_LENGTH xmk_TmpDataLength,
              void xmk_RAM_ptr  xmk_TmpDataPtr
            #endif

            #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
              xPID xmk_TmpReceiverPID
            #endif

   Return: xmk_OPT_INT
```

The function `xOutEnv()` exists as a template in the generated module `env.c`.

Each time an SDL output occurs, the generated C code calls the C function xmk_Send or xmk_SendSimple. After performing some checks, and if the signal is to be sent to the environment, the C function xOutEnv() is called with all the parameters necessary to represent the signal. The parameters are explained in the following.

With the parameter xmk_TmpSignalID, the signal ID is to be specified.

With the xmk_TmpPrio parameter, the signal's priority is to be specified (if conditionally compiled). If the XMK_USE_SIGNAL_PRIORITIES is not defined, the signal priorities which are specified with #PRIO in the diagrams is just ignored. The use of signal priorities is not recommended because this violates SDL. A few bytes can be spared if signal priority is not used. See also XMK_USE_SIGNAL_PRIORITIES.

With the parameter xmk_TmpDataLength, the number of bytes of signal parameters is to be specified. The number of bytes is evaluated by using a sizeof (C struct) construct. If the signal carries no parameters, this value is set to 0.

With the xmk_TmpDataPtr parameter, a pointer to the memory area containing the parameter bytes of the signal is given. The memory area is not treated as dynamically allocated within this function. Because the function copies the parameter bytes, the caller may use any temporary memory (for example memory allocated from the C stack by declaring a C variable). This parameter should be set to NULL if no parameter bytes are to be transferred (if conditionally compiled).

The parameters xmk_TmpDataLength and xmk_TmpDataPtr are compiled conditionally. The XMK_USED_SIGNAL_WITH_PARAMS is automatically generated into the sdl_cfg.h file, from the Cmicro SDL to C Compiler. For tiny systems, if there are no SDL signals with parameters specified, this is undefined. It will reduce the amount of information which is to be transferred for each signal, with a few bytes. See also XMK_USED_SIGNAL_WITH_PARAMS.

With the last parameter xmk_TmpReceiverPID, the PID of the receiving process is to be specified (if conditionally compiled). The parameters xmk_TmpDataLength and xmk_TmpDataPtr are compiled conditionally, see also XMK_USE_RECEIVER_PID_IN_SIGNAL.

If XMK_USE_RECEIVER_PID_IN_SIGNAL is not defined, the user must implement the C function xRouteSignal which is responsible to derive the receiver from the signal ID in that case. Using xRouteSignal

is recommended only if the last few bytes must be spared for transferring of signals.

Note:

There is a file called `<systemname>.ifc`, which is generated by the Cmicro SDL to C Compiler. It is necessary to include this file together with `ml_typ.h` before defining `xOutEnv()`. This is necessary to have access to all the objects that are generated by the Cmicro SDL to C Compiler. The user should also make sure that this file is generated, because the analyzer's make menu contains an option for this.

Evaluate the Signals's Sender

The sender of the signal can be retrieved by the global variable `xRunPID` of type `xPID`. Users should remember, that in a system with only (x,1) process declarations, this pid represents the process type of the sender. In a system in which multiple process instances of the same process type appear, this pid represents the process type plus the process instance of the sender. Probably it is required to make decisions depending on the sender of the signal. The user can evaluate the process type of the sender by using the C expression `EPIDTYPE(xRunPID)`, i.e.:

Example 576: Evaluating the Process Type

```
unsigned char ptype;
ptype = EPIDTYPE(xRunPID);
```

Remember also, if it is required to signal to a specific pid in the environment, this requires dynamic signalling where a process in the environment establishes communication to a process in the system or the other way around.

Checking the Signal's Receiver

Under normal circumstances it is not necessary to check the receiver in `xOutEnv()` as the Cmicro Kernel calls `xOutEnv()` only, if the environment is recognized as the signal's receiver.

Caution!

The users have to ensure that signals to the environment are consumed in the environment by returning `XMK_TRUE` in `xOutEnv()`.

Signal Parameters

If the signal to be sent to the environment contains parameters, it is necessary to copy these parameters to the data area of the environment as the signal and its parameters are deleted after signal consumption.

For each signal carrying parameters, there is a typedef struct generated into the `<systemname>.ifc` file. Please view Example 577.

Return Values of xOutEnv()

The user must ensure that signals which are to be sent to the environment are consumed by the environment. In `xOutEnv()` this is done by returning the values `XMK_TRUE` or `XMK_FALSE`.

- `XMK_TRUE`
  The signal has been consumed by the environment.
- `XMK_FALSE`
  The signal is NOT consumed by the environment.

An Easy Example

Assume, for example, a process which has to send a signal with parameters to the environment. The code the user has to write into the `xOutEnv()`-function, looks as follows:

Example 577: `xOutEnv()`

```
xmk_OPT_INT xOutEnv (xmk_T_SIGNAL      xmk_TmpSignalID,
                     xmk_T_PRIO        xmk_TmpPrio,
                     xmk_T_MESS_LENGTH xmk_TmpDataLength,
                     void *            xmk_TmpDataPtr,
                     xPID              xmk_TmpReceiverPID )

{
  xmk_OPT_INT xmk_TmpResult = XMK_FALSE;

  switch (xmk_TmpSignalID)
  {
    case SDL_Signal1 :
      {
        /* BEGIN User Code */
        int temp;
        temp = (yPDef_z4_SDL_Signal1*)xmk_TmpDataPtr->Param1
        UserFunction(temp);
        /*   END User Code */
        xmk_TmpResult = XMK_TRUE; /* signal is */
                                  /* consumed  */
      }
      break ;

   case SDL_Signal2 :
      {
        /* BEGIN User Code */
        char g;
        int  k;
        g = (yPDef_z5_SDL_Signal2*)xmk_TmpDataPtr->Param1;
        k = (yPDef_z5_SDL_Signal2*)xmk_TmpDataPtr->Param2;
        UserFunction2( g, k );
        /*   END User Code */
        xmk_TmpResult = XMK_TRUE; /* signal is */
                                  /* consumed  */
      }
      break ;

   default :
        xmk_TmpResult = XMK_FALSE;/* signal is NOT */
                                  /* consumed   */
                                  /* and to be handled */
                                  /* by the Cmicro Kernel */
        break ;
  }
  return( xmk_TmpResult );
}
```

Caution!

The function `xInEnv()` contained in the file `env.c` is generated by the Targeting Expert. To secure that changes done by the user will not be lost when generating again it is allowed only to modify this file between the comments `/* BEGIN User Code */` and `/* END User Code */`.

Furthermore it is not allowed to remove these comments or to add similar ones at other places.

Inter-Processor-Communication

> **Note:**
>
> If inter-processor-communication is to be performed, the user has to define a unique protocol between the communicating processors. In small applications with a restricted range, it might be enough to copy C structures, but usually this causes problems when it comes to redesigning the hardware.

Closing the Environment / the Interface to the Environment

xCloseEnv()

Parameters:

```
In/Out: -no-
Return: -no-
```

There is one C function called `xCloseEnv`() available as a template in the generated C module `env.c`.

The user should fill this function out appropriately thus realizing disconnection from the environment (drivers, interrupt service routines and so on). Disconnection can make sense if a re-initialization or a software reset is to be implemented.

The `SDL_Halt` function is mapped to `xCloseEnv()` in Cmicro.

SDL System Time Implementation

Included in the Cmicro Kernel, there are some template functions for the implementation of SDL system time. All these functions are contained in the module `mk_stim`. The functions `xmk_NOW()` or `xmk_SetTime()` must be implemented newly if a new method for accessing the hardware system time is to be implemented. In many cases, the standard C library function `time()` is available, which is used in most of the C compiler adaptations that are already been made.

It is necessary though to implement a function which makes the variable `SystemTime` topical. This will be an interrupt service routine in most cases.

Please view "Defining the SDL System Time Functions in mk_stim.c" on page 3435.

Getting the Receiver of a Signal – Using xRouteSignal

Parameters:

```
In/Out: xmk_T_SIGNAL sig
Return: xPID
```

The user can remove the receiver's xPID from the signal structure by resetting the flag "XMK_USE_RECEIVER_PID_IN_SIGNAL" on page 3398. In this case, the function `xRouteSignal()` has to be filled as the Cmicro Kernel needs to know which process likes to receive the current signal.

Example 578: Function `xRouteSignal()`

```
/*
** the <p> below stands for the automatically generated prefix
*/

xPID xRouteSignal  ( xmk_T_SIGNAL sig )
{
   /*
   ** Please insert appropriate code here to map
   ** Signal ID's to Process - Type - ID's
   ** (XPTID_ProcessName in generated code).
   ** Keep in mind that this function might be
   ** called from within a critical path.
   ** Include <systemname>.ifc to get signal names
   ** and process' XPTID
   */
   switch (sig)
   {
    /*
    **  S D L   T i m e r s ...
    */
    case SDL_Timer1: return (XPTID_<p>_ProcessName_A)
                     break;
    case SDL_Timer2: return (XPTID_<p>_ProcessName_B)
                     break;
    case SDL_Timer3: return (XPTID_<p>_ProcessName_A)
                     break;
    case SDL_TimerN: return (XPTID_<p>_ProcessName_C)
                     break;

    /*
    **  O r d i n a r y  S D L  S i g n a l s ...
    */
    case SDL_Signal1: return (XPTID_<p>_ProcessName_B)
                      break;
    case SDL_Signal2: return (XPTID_<p>_ProcessName_A)
                      break;
    case SDL_Signal3: return (XPTID_<p>_ProcessName_A)
                      break;
             ......

    case SDL_SignalN : return (XPTID_<p>_ProcessName_C)
                      break;

    default: ErrorHandler (ERR_N_NO_RCV);
             return (xNULLPID)
             break;
   }
}
```

Dynamic Memory Allocation

General

Dynamic memory allocation in real life always introduces problems, which are:

- The memory occupation cannot be evaluated by the user, so that it is impossible to configure the dynamic memory. If all objects are allocated statically and if the memory occupation exceeds the available memory, this results in errors during compilation/linking, or at least a memory map file can be viewed.
- If dynamic memory allocation is used, then, after the program has executed for a time, usually memory leaks are the result. Memory leaks are fatal because there might be enough memory, but it is even impossible to allocate one more block.
- If there is no more free dynamic memory available, then it is up to the user to decide how to continue in the program. In any case, the program should be terminated, started again or any similar reaction is to be programmed.

Normally, Cmicro tries to prevent any use of dynamic memory allocation, but there are the following exceptions, in which this is impossible:

- When a signal with too many parameters is to be sent to another process. See "Signals, Timers and Start-Up Signals" on page 3368.
- When an SDL sort is used requiring dynamic memory management, like the charstring sort.
- If the SDL Target Tester is used, because there is dynamic memory allocation in the start-up phase and for each transmit buffer.

There are at least two possibilities to implement dynamic memory allocation namely:

- The dynamic memory management from the C Compiler or operating system can be used.
- The dynamic memory management from Cmicro can be used.

When it comes to targeting, the user decides upon the dynamic memory allocation manager. In the following, the both possibilities are explained.

Dynamic Memory Allocation Functions – Compiler or Operating System

If the C compiler, that the user is using in the target environment, provides dynamic memory allocation functions, it is possible to use these functions. A few steps must be carried out to include the dynamic memory allocation functions of the C compiler or operating system, which are:

- The user must introduce his own C compiler section, please refer to the subsection "Introducing a new C Compiler" on page 3432.
- In the new C compiler section, the correct header files must be introduced with `#include`. The user should refer to the manuals of the C compiler or operating system.
- The SDL Target Tester, Cmicro Library and Cmicro Kernel always allocate dynamic memory by using the functions in the file `mk_cpu.c`.
- The file `mk_cpu.c` below the Cmicro template directory must be copied into the user's project directory. Any modifications should be performed on the private copy of `mk_cpu.c`.
- The `mk_cpu.c` file contains two C functions called `xAlloc()` and `xFree()`. Within these functions, the user should call the appropriate dynamic memory allocation functions of the C compiler or operating system.

Dynamic Memory Allocation Functions – Cmicro

Below the Cmicro kernel directory there is a C module `ml_mem.c` that implements a dynamic memory allocator. The C module `ml_mem.c` can be used for managing memory dynamically for SDL systems, but it is not forbidden to use this module within handwritten C code also.

The module cannot be used if partitioning is to be used. In that case compilation errors will occur. Please refer to "Dynamic Memory Allocation" on page 3450 for an explanation which parts in SDL are to be dynamically allocated. The module provides C functions for initialization, allocation, de-allocation, and getting some information about the current memory status.

There is only one memory pool. The memory pool is to be declared by the user and initialized with the C function `xmk_MemInit` before the

memory pool can be used. Allocations may be performed by calling `xmk_Malloc()` or `xmk_Calloc()`. An allocated block is de-allocated again by calling `xmk_Free()`. So far, the principle behavior is the same as the usual `malloc()`, `calloc()` and `free()` functions from compilers. But the memory pool can probably be cleaned with a Cmicro specific function. There are additional functions which can be used to query the amount of free or occupied space.

Before the memory management functions of Cmicro can be used, a few adaptations are to be made which are explained in the following.

Adaptations That Are to Be Made

Some adaptations are to be made by the user, which are necessary to include the right functions, scale buffers and memory management and take care for general adjusting like alignment of the CPU.

- For general use of the Cmicro memory management functions the user should set the flag `XMK_USE_SDL_MEM` in `ml_mcf.h` with the help of the Targeting Expert. This will make the basic memory allocation functions available. The `ml_mem.c` module must of course be compiled an linked together with the application.
- Adjust the alignment that the CPU is using. There is a macro definition in `ml_mem.c` called `CPU_WORD_SIZE`. The right setting of this macro is basically important for making the dynamic memory allocation functions do work. With this flag, it is possible to define the word size of the target CPU. The value is predefined in `ml_mem.c` but it could be the case that the predefined value is inappropriate for the target system. If the `CPU_WORD_SIZE` value must be redefined, this could be done in `ml_mcf.h` in the user's section. The predefined value for UNIX compilers is 8, the predefined value for `ARM_THUMB` C compiler 4, otherwise a default value of 1 (no alignment) is used.
- Here is a recommendation: The user should use an alignment of 4 (32 Bit) if it is not sure what type of alignment the C compiler / CPU produces. This will work for most cases, but of course probably not if the CPU is 64 Bit CPU. For small systems, this might not be appropriate, because there is an extra overhead of 3 bytes per allocated block. For a CPU like `8051` and derivatives a `CPU_WORD_SIZE` of 1 is appropriate.

- It is possible to introduce a minimum block size per each allocated block. This decreases the risk of getting memory leaks after a while because blocks of the same size can be put together again. If the mechanism of a minimum block size is to be used, then the define XMK_USE_MIN_BLKSIZE is to be used. With the define XMK_MEM_MIN_BLKSIZE the user may define the minimum block size per each allocated block. If the XMK_MEM_MIN_BLKSIZE is not defined from the user, a minimum block size of 64 is predefined.
- The ml_mem.c module contains profiler that keeps track on how many blocks are allocated, how much memory is free and furthermore. The user must define XMK_ADD_PROFILE in order to get the complete functionality. If the SDL Target Tester is used, the flag is automatically predefined.

xmk_CleanPool()
Parameters:
```
In/Out: -no-
Return: size_t
```
This C function returns the amount of occupied memory. It is available only if XMK_USE_SDL_MEM and XMK_SYSTEM_INFO are set.

xmk_GetOccupiedMem()
Parameters:
```
In/Out: -no-
Return: size_t
```
This C function returns the net amount of occupied memory. It is available only if XMK_USE_SDL_MEM and XMK_SYSTEM_INFO are set.

xmk_GetFreeMem()
Parameters:
```
In/Out: -no-
Return: size_t
```
Returns the amount of free memory in sum, which means that the overhead from each block is included. It is available only if XMK_USE_SDL_MEM and XMK_SYSTEM_INFO are set.

xmk_EvaluateExp2Size()
Parameters:
```
In/Out: size_t
Return: size_t
```

This function is either used from Cmicro's dynamic memory allocation functions `xmk_Malloc()` and `xmk_Calloc()` or it may be used directly from the user.

It is available only if `XMK_USE_MIN_BLKSIZE` is set.

The function is declared as:

```
size_t xmk_EvaluateExp2Size ( size_t GivenLength )
```

The function evaluates from the given length a length value, which is in any case a 2 exp N value. This is used to reduce the risk of memory leaks that occur in dynamic memory management systems. It may be used for the memory functions of this module but also for the memory functions of an operating system or C compiler. If the minimum block size is in any case greater than the greatest block that is requested in the target system, then there is no risk for memory leaks. The return result is either the minimum specified with `XMK_MEM_MIN_BLKSIZE`, but might be also one of the following values:

```
64,128,256,512,1024,2048,4096,8192,16384,32768,65536
```

The ?Memory Command

The SDL Target Tester offers a command that allows the user to inspect the status of the dynamic memory. Unfortunately this command can be used only, if the dynamic memory management from Cmicro is used (by setting the `XMK_USE_SDL_MEM` flag in the Targeting Expert).

The command allows the user to check the following:

- Is the dynamic memory correctly initialized after system start-up?
- How is the dynamic memory pool configured?
- At any time during SDL target system execution:
  - what was the greatest block that was allocated?
  - how many blocks are currently in the pool?
  - what is the maximum amount of blocks that have ever been in the pool?
  - physical address of the memory pool (the address of the
  - first block of structure `xmk_T_MBLOCK` (see `ml_mem.c`)

The command presents the following output to the user:

```
M-STATE:Memory pool size incl.overhead     =4096
M-STATE:Current memory fill                =136
M-STATE:Current amount of blocks in pool   =2
M-STATE:Peak hold: Amount of blocks        =29
M-STATE:Peak hold: Largest block           =2048
M-STATE:Overhead per block (in bytes)      =20
```

```
M-STATE:Minimum block size                =0
M-STATE:Memory pool address (hex)         =28478
M-STATE:(Probably there are memory leaks)
```

The command is available only if XMK_USE_SDL_MEM and XMK_ADD_PROFILE are set.

User Defined Actions for System Errors – the ErrorHandler

Parameters:

```
In/Out: int ErrorNo
Return: void
```

Errors, warnings and information can be generated and detected in many places and situations during the lifetime of an SDL system. By fully utilizing the Analyzer, several dynamic error sources can be eliminated at the design stage of development.

Some errors or warnings go undetected by the Analyzer, for example resource errors or real time errors such as memory, performance, or illogical use of SDL.

These errors and warnings are detectable by the Cmicro Kernel and the SDL Target Tester. For a complete list of errors, warnings and information, please view the following subsection "List of Dynamic Errors and Warnings" on page 3457.

As a general rule, for each error and warning there should be an appropriate reaction in the function ErrorHandler() in the mk_user module.

The ErrorHandler() is, as a default implementation, implemented like this:

In the case of a warning message, either:

- The actions defined via the flag XMK_WARN_ACTION_HANG_UP are carried out (see "Reactions on Warnings" on page 3403).
- The actions defined via the flag XMK_WARN_ACTION_PRINTF are carried out.
- The actions defined via the flag XMK_WARN_ACTION_USER and XMK_WARN_USER_FUNCTION are carried out.

In the case of an error message, either:

- The actions defined via the flag XMK_ERR_ACTION_HANG_UP are carried out (see "Reaction on Errors" on page 3405)

- The actions defined via the flag `XMK_ERR_ACTION_PRINTF` are carried out.
- The actions defined via the flag `XMK_ERR_ACTION_USER` and `XMK_ERR_USER_FUNCTION` are carried out.

For more explanations on these flags, please view "Reactions on Warnings" on page 3403 and "Reaction on Errors" on page 3405. If `printf` is used, the C module `ml_mon.c` below the `cmicro/kernel` directory must be compiled also.

Errors are numbered easily by an integer value. If the user's target allows the use of `printf`, then use the C function `xmk_err_text` contained in the `ml_err` module. It is easier to specify error handling in this module rather than directly modifying the `ErrorHandler` function.

Example 579: `Default implementation of ErrorHandler()` ———

```
void ErrorHandler ( int ErrorNo )
{
  ...
  ...
  ...

 ErrorClass = xmk_GetErrorClass (xmk_TmpErrorNumber);
  #ifdef XMK_WARN_ACTION_HANGUP
    if (ErrorClass==XMK_WARNING_CLASS)
      while (1);
  #endif /* ... XMK_WARN_ACTION_HANGUP */

  #ifdef XMK_ERR_ACTION_HANGUP
    if (ErrorClass==XMK_FATAL_CLASS)
      while (1);
  #endif /* ... XMK_ERR_ACTION_HANGUP */

  #ifdef XMK_WARN_ACTION_PRINTF
    if (ErrorClass==XMK_WARNING_CLASS)
      xmk_MonError (stderr, xmk_TmpErrorNumber);
  #endif /* ... XMK_WARN_ACTION_HANGUP */

  #ifdef XMK_ERR_ACTION_PRINTF
    if (ErrorClass==XMK_FATAL_CLASS)
      xmk_MonError (stderr, xmk_TmpErrorNumber);
  #endif /* ... XMK_ERR_ACTION_HANGUP */

  #ifdef XMK_WARN_ACTION_USER
    if (ErrorClass==XMK_WARNING_CLASS)
      XMK_WARN_USER_FUNCTION( xmk_TmpErrorNumber );
  #endif /* ... XMK_WARN_ACTION_USER */

  #ifdef XMK_ERR_ACTION_USER
    if (ErrorClass==XMK_FATAL_CLASS)
      XMK_ERR_USER_FUNCTION( xmk_TmpErrorNumber );
  #endif /* ... XMK_ERR_ACTION_USER */

  ...
  ...
  ...
```

```
} /* END OF SAMPLE */
```

Example 580: `ErrorHandler()`

```
void ErrorHandler ( int ErrorNo )
{
  /* The user could implement the 3 functions
  ** fatalerror, warning, and information
  */
  switch (ErrNo)
  {
    case  ERR_N_UNKNOWN
          fatalerror(ErrNo);
          break;

    case  .....
          .....
          break;

    default: warning(ERR_N_UNKNOWN); break;
  }
} /* END OF SAMPLE */
```

List of Dynamic Errors and Warnings

Cmicro Kernel Errors

- ERR_N_CREATE_NO_MEM

  This error can happen in the case of using dynamic process creation. When a parent process tries to create another process, the Cmicro Kernel tries to allocate an entry in the queue for the insertion of an internal create signal.

  The error occurs when full queue capacity has been reached.

  Help: Change the size of the signal queue by modifying the macro XMK_MAX_SIGNALS in `ml_mcf.h`.

- ERR_N_CREATE_INSTANCE

  This error happens if there is no dormant instance of a process type which can be (re)used. The Cmicro Package uses fixed upper process instance limits (x, N), where N is to be specified as an integer value. In the C code, all the data of one process instance is represented by one element of an array of the size N.

The error occurs if all instances (i.e. all array elements), of this process type are currently active and the parent tries to create another instance of that type.

- ERR_N_SDL_DISCARD

  A signal was sent to a process not existing (either not yet created or already stopped). The signal is therefore discarded.

- ERR_N_DIVIDE_BY_ZERO

  A division by zero has been detected in the generated SDL code, which is an SDL user error. Possibly, it is helpful to produce an execution trace to localize the error. Normally, it is possible to find such problems within the simulation.

- ERR_N_NO_FREE_SIGNAL

  It is impossible to allocate one more signal instance from the static memory pool. Depending on how the user has defined the signal handling (`XMK_USE_STATIC_QUEUE_ONLY` and `XMK_USE_STATIC_AND_DYNAMIC_QUEUE`), this must be treated either as a warning or as a fatal error.

  It has to be treated as a fatal error, when the macro `XMK_USE_STATIC_QUEUE_ONLY` is set.

  It can be treated as a minor warning, when the macro `XMK_USE_STATIC_AND_DYNAMIC_QUEUE` is set.

  The output operation fails, if it is impossible to allocate one more signal.

  Help: Change the size of the signal queue by modifying the macro XMK_MAX_SIGNALS in `ml_mcf.h` or change the size of predefined dynamic memory pool (the memory pool, that is used when the xAlloc C function is called).

- ERR_N_PARAMETER_MEM_ALLOC

  The output operation fails as a signal with parameters is to be sent but not enough free memory is available for allocation of the signal parameters.

  If the signal parameter's length is greater than the value of XMK_MSG_BORDER_LEN (see "Compilation Flags" on page 3394) the parameters are inserted in a memory area which has to be

allocated. This allocation fails because there is no more memory available.

Help: If using the Cmicro Library's memory functions (flag XMK_USE_SDL_MEM): The size of the memory for allocation can be increased by incrementing the value XMK_MAX_MALLOC_SIZE.

- ERR_N_SYSTEM_SIGNAL

  This error can only happen, if a non specified system signal is sent. Internally, some signals are predefined as system signals. System signals are given priority treatment. For example, for the dynamic process creation, there is a system signal called `XMK_CREATE_SIGNALID` which is automatically defined in `ml_mcf.h` according to the setting of XMK_USE_MORE_THAN_250_SIGNALS.

  To prevent this error situation happening it is strongly recommended not to send an SDL signal with equal or higher priority than that of a system signal (please see the subsection "Scheduling" on page 3377).

- ERR_N_NO_CONT_PRIO

  This error occurs when process priorities are not numbered according to the rules of the preemptive Cmicro Kernel. The numbering must begin from zero incremented consecutively to an upper limit. All numbers between zero and the upper limit-1 are to be used within a #PRIO directive, no number may be omitted. (See MAX_PRIO_LEVELS and `xDefaultPrioProcess` in section "Preemption" on page 3409.

- ERR_N_NO_COR_PRIO

  When the preemptive Cmicro Kernel is selected, the macro MAX_PRIO_LEVELS has to be set correct. As described in MAX_PRIO_LEVELS in section "Preemption" on page 3409, this macro has to contain the amount of process priority levels.

- ERR_N_xRouteSignal

  The function `xRouteSignal`, which is to be filled by the user when signals do not contain a receiver pid, detects an error or was not able

to handle the current signal which is to be routed. See "Bare Integration" on page 3437.

- ERR_N_NO_REC_AVAIL

  If the user is using implicit addressing (output without to) and there are several possible receivers of one type, the Cmicro Kernel tries to assign one of them to this signal. This assignment may fail if there is no possible receiver. If there is more than one, then the first one found will be used in the output of the C function `xmk_Send*`.

- ERR_N_NO_FREE_TIMER

  The error occurs when an SDL process tries to start an instance of a timer and either the timer is unknown or there is no memory available to start a timer instance. To eliminate the first source of error check the SDL compilation. For the second case increase the memory assigned to timers by increasing the value of XMK_TIMERPRIO.

- ERR_N_PID_INDEX

  The Cmicro Kernel uses PIds as index values. Each SDL process in the generated system is numbered by a system-wide unique number. There is an error detected when the index is out of range. This problem most probably lies in the environment which tries to send a signal to a non existing process.

- ERR_N_SEND_TO_NULLPID

  An SDL application tries to send to a NULL - PID. This case normally cannot arise as the Analyzer performs appropriate checks in the dynamic analyses pass. If it occurs, it is most probably that either something is wrong with the environment signalling or possibly with initialization of pid variables.

- ERR_N_TRANS_TIME

  The maximum execution time for one SDL transition was exceeded. This error can only happen if no preemption is used and if the administration of the transition-execution-time is switched on. See "XMK_USE_CHECK_TRANS_TIME" on page 3408.

- ERR_N_xOutEnv

The C function `xOutEnv()` does not handle all necessary signals. It should handle all signals which are sent to the environment. Please view "Return Values of xOutEnv()" on page 3446.

- ERR_N_INIT_SDL_MEM

  If the Cmicro Library's memory functions are used, (XMK_USE_SDL_MEM is defined) the memory for `alloc()` and `malloc()` has to be initialized. This error occurs if a `malloc()` takes place before the memory is initialized. See "ml_mem.c" on page 3473

- ERR_N_SDL_DECISION_ELSE

  An SDL decision without any ELSE branch leads to a fatal error.

- ERR_N_SDL_RANGE

  The index of an SDL array has crossed the range of the SDL array. The definition of the array has to be checked within SDL.

- ERR_N_NO_RCV

  If the flag XMK_USE_RECEIVER_PID_IN_SIGNAL is undefined the function `xRouteSignal()` has to be filled by the user. See "Getting the Receiver of a Signal – Using xRouteSignal" on page 3449. This error code means that there is no receiver defined for the current signal.

- ERR_N_SDL_IMPLICIT_CONSUMPTION

  The receiver process is not expecting the current signal in his current state, so the signal was implicitly consumed.

- ERR_N_PREDEFINED_OPERATOR_CALL

  An error occurred in the call to one of the predefined operators. Unfortunately it is not possible to find out what the reason for this error is without either simulating the system or using the SDL Target Tester or any trace possibility introduced by the user. The error message occurs if a range in the memory is crossed.

- ERR_N_INIT_QUEUE

  The SDL signal queue is not correctly initialized. The error occurs if the user has forgotten to call the C function `xmk_InitQueue()`. This function must be called before `xmk_InitSDL()` may be called.

Please view the section "Implementation of Main Function" on page 3437.

- ERR_N_MEM_PARAM

  This error occurs if the user tries to call to the `xmk_Malloc()` C function with a wrong parameter. It is not allowed to request a memory block of the size 0.

- ERR_N_MEM_NO_FREE

  There is no free block in the memory pool of the Cmicro Memory Management (see "Exported from ml_mem.c" on page 3494). This error can only be detected if the Cmicro Memory Management is used (see the flag "XMK_USE_SDL_MEM" on page 3422).

  The size of the dynamic memory, i.e. the amount of blocks available can be modified by modifying the flag XMK_MAX_MALLOC_SIZE

- ERR_N_MEM_ILLMBLOCK

  A call to the `xmk_Free()` function of the Cmicro Memory Management (see the file "Exported from ml_mem.c" on page 3494) occurred and the given block is an invalid block.

  This error can only be detected if the Cmicro Memory Management is used (see the flag "XMK_USE_SDL_MEM" on page 3422).

- ERR_N_NO_FREE_SIGNAL_DYN

  This error is detected when a new signal is to be allocated dynamically and there is no more free memory available. The Cmicro Kernel starts to create signal instances by using memory allocation in the case that the define XMK_USE_STATIC_AND_DYNAMIC_QUEUE is defined and there is no available signal instance in the predefined pool of static signal instances (see XMK_MAX_SIGNALS).

- ERR_N_NO_FREE_TIMER_DYN

  This error is detected when memory for a new timer instance is to be allocated dynamically and there is no more free memory available. The error only occurs if timers with parameters are used. The Cmicro Kernel starts to create timer instances by using memory allocation in the case that timers with parameters are used in SDL

(XMK_USED_TIMER_WITH_PARAMS is defined in `sdl_cfg.h`) and the amount of statically predefined timer instances (XMK_MAX_TIMER_USER defined with Targeting Expert) is being reached.

- ERR_N_NULL_POINTER_VALUE_USED

  This error occurs within the macros that are generated in order to check null pointer access. These error checks are generated for variables of sort ref, own and oref. If the error occurs, the further behavior of the SDL system is unpredictable.

- ERR_N_UNDEFINED_ERROR

  This error message is implemented to serve the C `switch` statement in use with a default branch. It should not occur during system execution.

SDL Target Tester Errors

- ERR_N_INDEX_SIGNAL

  Under normal circumstances this error should not occur. It is an error detected by the SDL Target Tester when a signal id is out of range. This is only used if signal trace options are modified or asked for.

- ERR_N_ILLEGAL_CMD

  An illegal command was sent to the SDL Target Tester command interface module. This error situation arises due to careless implementation of the function interface and should not occur under normal circumstances.

- ERR_N_TESTER_MESSAGE

  An illegal message was sent to the SDL Target Tester and it is not able to decode the message. A possible error source is an inconsistency in the underlaying protocol on the communications interface.

- ERR_N_LINK_SYNC

  It was not possible to receive the sync byte of the data link frame of the Cmicro Protocol.

- ERR_N_LINK_DEST_BUFFER

A message which is to be decoded is too large for the destination buffer. Possible error sources are an inconsistency in the data definition or the length of information received is wrong.

- ERR_N_LINK_ENC_LENGTH

  An attempt was made to send more than the allowed maximum amount of bytes.

- ERR_N_LINK_ENC_MEMORY

  There is not enough, or no free memory to encode a data link frame of the Cmicro Protocol.

- ERR_N_LINK_NOT_IMPL

  A feature of the data link which is not implemented has been used, for example an attempt to send more than the maximum amount of allowed bytes.

- ERR_N_DATA_LINK

  There is an error on the data link detected.

- ERR_N_DECODE_METHOD

  There is no decoding method defined for a given frame of the Cmicro Protocol.

- ERR_N_RING_WRITE_LENGTH

  There is a ring buffer overflow detected in the data link module.

- ERR_N_TRACE_OUTPUT

  The signal parameter length is greater than the size which can be transmitted.

- ERR_N_RECORD_OUTPUT

  The signal parameter length is greater than the size which can be transmitted.

- ERR_N_RECORD_MAX_EXCEEDED

  The debit counter used for the SDL Target Tester's record is overflown.

- ERR_N_RECORD_STATE

  The received message cannot be handled in the current recorder state.

- ERR_N_RECORD_CANNOT_CONT

  Because a fatal system error is detected the record session cannot be continued.

Light Integration

A light integration is to be performed if the SDL system including the Cmicro Library should execute within just one operating system task.

The SDL system usually communicates with the environment by using a mailbox or message queue or something similar, which is provided from the operating system. Within the SDL system, the Cmicro scheduler is used. This means that signals that are sent internally in the SDL system are not sent via the mailbox/message queue from the operating system.

Model

The execution model for an SDL system in a light integration can be sketched with the following meta code:

1.

```
/* In the user's initialization task do : */
Initialize memory
Initialize time
Initialize mailbox(es) or message queue(s)
Initialize other resources
Make the SDLTask an operating system task
Wait until the SDLTask has finished initialization
Continue with creating other operating system tasks
```

The model for what the SDL task does in a light integration can be sketched with something like:

2.

```
Initialize SDL Target Tester
Initialize Cmicro SDL queue
Initialize Cmicro Trace options (if wanted)
Initialize SDL and execute start transitions of the
process instances which are to be created statically

Forever do
{
  if there is a signal to be consumed
  {
    Get the next message from the OS queue,
    without suspending the SDL Task
  }
  else, if there are only saved signals
  {
    evaluate the remaining duration for the
```

```
      timer that will expire next

      if there is at least one active timer
      {
        listen on the operating system queue for the
        remaining duration, by using a blocking
        function call
      }
      else
      {
        listen forever to the operating system queue
        by using a blocking function call
      }
      If a message was received, format the message
      and send it as an SDL signal to the SDL system
    }
    If a timeout occured (remaining duration),
    then check all the timers. For each timer that
    has been expired there will be one signal
    put into the SDL queue

    process SDL signal
  }
```

Procedure to Implement the Model

This subsection gives the user the details that he must know for the implementation of the given model. He has to deal with some special macros which must be defined and used. Their names have the prefix `XLI_`.

The user must execute the following steps:

1. Create the module with the C `main()` function.

This module has to contain the C `main()` function, corresponding to the first meta code in the section before. The module should include `ml_typ.h` and the interface file (e.g `component.ifc)`. In this module the definitions of other external tasks can take place. The main function should handle the following steps:

- initialize the memory if required
- initialize the hardware timer
- initialize OS resources (semaphore, message queues,...)
- start the necessary OS tasks, this means also to start the task which represents the SDL state machine.
- wait until all tasks are started

2. Set XLI_LIGHT_INTEGRATION

To set this flag the Targeting Expert can be used. Choose your integration and choose “Support light integration” below Target Library and kernel tab. When the box is chosen three things will happen.

- the compiler flag XLI_LIGHT_INTEGRATION will be set
- the compiler flag XMK_USE_INTERNAL_QUEUE_HANDLING is automatically set
- The macro XLI_INCLUDE is set with the name in the edit line (default is ’li_os_def.h’)

This header file is included in ml_typ.h and must contain a definition of XLI_SDL_TASK_FUNCTION. With the help of this macro, the main function known from the standard Cmicro kernel will be mapped to the OS task function. The body of this function is located in mk_user.c.

3. Define the macros which are needed in the task function.

The task function is corresponding to the second meta code of the model. The following macros can be defined in the header file:

- XLI_TEMP_TASK_VARS

If temporary variables are needed in the task function, they can be declared at this place. It is inserted at the top of the task function.

- XLI_TEMP_QUEUE_VARS

If temporary variables for the communication resources are needed, they can be defined within this macro. It is expanded at the top of the function which queries the queue.

- XLI_OS_TASK_INIT

Any preparations in the task function can be done with this. It follows XLI_TEMP_TASK_VARS at the start of the task function.

- XLI_CREATE_OS_QUEUE

Here the communication resources for the OS task can be created (which ones that will be used depends on the user and the used OS). In the further part of the document the term queue will be used instead of communication resources.

- XLI_START_OS_TASK_SYNC

If a synchronization between the OS tasks is needed, this should be defined with XLI_START_OS_TASK_SYNC.

- XLI_GET_NEXT_MESSAGE_FROM_OS_QUEUE

The function xmk_RunSDL() represents the continuous repeating part of the meta model. Inside of it xmk_QueryOSQueue() is called and it is responsible for the query of the OS queue. If the SDL queue contains signals, XLI_GET_NEXT_MESSAGE_FROM_OS_QUEUE is used to read non-blocking from the OS queue. The task function should not be suspended.

- XLI_LISTEN_ON_OS_QUEUE_WITH_TIMEOUT(TIMEOUT)

If the SDL queue is empty, but at least one timer is active, XLI_LISTEN_ON_OS_QUEUE_WITH_TIMEOUT(TIMEOUT) is used to read from the OS queue with a time-out. The OS queue should be queried as long as the next timer expire. The duration is given with the parameter TIMEOUT.

- XLI_LISTEN_ON_OS_QUEUE_NO_TIMEOUT

If there is no signal in the SDL queue and there is no timer active XLI_LISTEN_ON_OS_QUEUE_NO_TIMEOUT is used to query the OS queue while the next OS message arrives. The SDL task should be suspended.

- XLI_CALL_xInEnv

The content of XLI_CALL_xInEnv is executed after an OS message arrives. It should pass on the content of the received OS message (a signal with parameter) to the SDL system. Normally xInEnv is called to send the signal to the SDL system. The user has to define a data structure which transports the signals from the OS to the SDL system. The name for the data structure can be assigned with XMK_xInEnv_PARTYPE. The name for the parameter of xInEnv can be assigned with XMK_xInEnv_PARNAME.

- XLI_END_OS_TASK_SYNC

This can be defined with something that sign up the other non-SDL tasks about ending of the SDL task.

4. Initialize the environment.

This is usually done in `xInitEnv()`, but probably outside SDL, depending on the needs of the user.

5. Adaptions for sending signals from the SDL task to any other OS task.

Usually, users must fill out the function `xOutEnv()` for sending signals to the environment (another operating system task). It is up to the user what kind of communication is to be used in this direction.

Note:

All necessary resources, e.g. the receivers queue, must have been created before they are used.

6. Adaptions for receiving signals from any other OS task and translation of the signal parameters.

In the SDL task, it is necessary to format the incoming parameters after the message was read from the message queue. Afterwards the signal can be sent to the SDL system with `XMK_SEND_ENV`.

Note:

All necessary resources must have been created before they are used.

7. Implement timers in SDL.

It should be possible to implement hardware timers, like it is described in "Defining the SDL System Time Functions in mk_stim.c" on page 3435. A usual way is to increment a global variable (of type `xmk_T_TIME`) by using an operating system task or similar. A time overrun is automatically detected by the Cmicro Kernel.

8. System shutdown.

There is no default way to shutdown, but usually it is enough to implement the `xCloseEnv` function and call it at an appropriate place.

9. Cmicro Target Tester

The next thing that is remaining is the SDL Target Tester. The instructions which have been given for bare integration, are still valid. The easiest way to use the SDL Target Tester in a light integration is to dedicate a communication interface exclusively for the SDL Target Tester.

Note:

The Target Tester can only used for tracing.

10. Compile and link

The added files must be registered as source files in the Targeting Expert. Please view "Source Files" on page 2861 in chapter 60, *The Targeting Expert* to get information on how to add more files to the list of files to be compiled. The Targeting Expert will automatically add these files to the makefile.

11. Save the settings and press Full Make.

File Structure

Description of Files

The Cmicro Library Functions and Definitions

sdl_cfg.h

This file is automatically generated by the Cmicro SDL to C Compiler into the directory which is currently active. It contains compilation flags used for the automatic scaling of the Cmicro Library and the generated C code. The file must not be edited by the user.

Caution!

The file `sdl_cfg.h` always carries the same name, for each SDL system generated and is stored in the currently active directory (project or working directory). Inconsistencies arise if several systems are to be generated in the same directory. To avoid this situation, it is recommended to use different working directories for each SDL system. Otherwise, unpredictable results at run-time could result, as some required automatic scalable features may/may not have been compiled.

ml_typ.h

This file is the central header file in the Cmicro Package. It contains

- more #includes
- defines, which are of global interest and Cmicro Library internal defines
- typedefs which are of global interest and Cmicro Library internal typedefs
- external declarations which are of global interest and Cmicro Library internal external declarations

sctpredg.h, sctpred.h and sctpred.c

These files contain all definitions necessary to handle SDL data, for example predefined sorts. `sctpred.h` is included in `ml_typ.h`.

ml_err.h

This header file defines all error numbers used by the Cmicro Kernel, the Cmicro Library and the SDL Target Tester.

ml_mem.c

This file contains the dynamic memory management functions from Cmicro. It contains among other functions the C functions `xmk_Malloc()`, `xmk_Calloc()` and `xmk_Free()`. Possible reasons to use this module are:

- Compiler does not support dynamic memory management.
- Dynamic memory management of the compiler does not meet the requirements of the application. This may be the case, if the user wants to use “best fit” instead of “first fit”, the first of which is normally not supported. It is possible to adapt the memory management to ones needs.
- User wants to use other mechanisms of `ml_mem.c`, like profiling. Please view “Dynamic Memory Allocation” on page 3450.

ml_mon.inc

This file is included by the `ml_mon.c` file. It exists only for internal purposes and in order to maintain all the defined errors in the system in a better way.

ml_mon.c

This file contains some help functions which are useful in producing screen outputs. It contains C functions for buffer printouts, SDL PId printouts, displaying error messages, and the display of the current scalings used in the executable. The file should usually be included when compiling the kernel.

ml_*.h

Other header files which contain extern declarations of modules of the Cmicro Library.

The Cmicro Kernel

mk_main.c

This file represents the main-interface to the SDL user. It contains functions which are to be called by the SDL user to integrate the Cmicro Kernel in his application.

- an SDL initialization function `xmk_InitSDL()`
- an SDL system execution function `xmk_RunSDL()`

For SDL Target Tester, there are some more functions to be called during initialization. Please view chapter 68, *The SDL Target Tester*.

> **Note:**
> The SDL queue must in any case be initialized before the SDL system is going to execute. The initialization function is called `xmk_InitQueue()` and is exported from the `mk_queu.c` module.

mk_user.c

This module is not in the Kernel directory, but in the Template directory. It contains function templates or examples which are to be filled out by the user:

```
main()
```

The C main function contains by default the full access to all Cmicro features.

```
ErrorHandler()
```

Central error handling routine is called each time an errors occurs.

```
WatchdogTrigger()
```

Handling of a hardware watchdog.

In earlier versions of the Cmicro Package the functions `xInitEnv()`, `xInEnv()`, `xOutEnv()` and `xCloseEnv()` were include in `mk_user.c`, too. These functions will now be generated by the Targeting Expert and stored in the file `env.c`.

mk_sche.c

This file is the heart of the Cmicro Kernel. It exports those functions which are used in the `mk_main` module and it uses those functions of

other modules which represent the SDL model. The module serves with all the different scheduling policies described later in the subsection "Scheduling" on page 3365.

mk_outp.c

This file contains the SDL operation OUTPUT which is represented by a few functions. There is a C function `xmk_Send()` representing the SDL output operation. In addition, there is a C function `xmk_SendSimple()` which is used when a signal contains no parameters and has no explicitly defined priority. This results in a more compact argument list thus reducing the generated C Code.

mk_queu.c

This module contains the data type "SDL queue" and defines all operations on the SDL queue. The queue handling covers all the aspects of the SDL semantics as far as signals are concerned. There is one function `xmk_InitQueue()` which must be called in the user's `main()` function before the SDL system is going to execute.

mk_tim1.c

This file contains the SDL operations on timers such as set, reset, active, and some help functions used by the Cmicro Kernel. The timer model is described within the subsection "Timers and Operations on Timers" on page 3375. The Cmicro Kernel has to initialize all timers, test for expired timers and reset all timers of a process, when a process instance stops.

mk_stim.c

This file contains some functions which are to be filled up by the user. This is the reason why it is in the Template directory. The functions are used by the Cmicro Kernel to get the system time used in SDL. The contents of this file should be seen as a template. No provision for the connection to hardware timers is provided in the delivered source, as the timer used is application dependent. The user is required to fill out the appropriate functions for the provision of SDL system time. Please view "Defining the SDL System Time Functions in mk_stim.c" on page 3435

mk_cpu.c

This file also contains some hardware specific functions which should be seen as templates. It is also in the Template directory.

mk_*.h

Other header files contain extern declarations of modules of the Cmicro Library. The contents and details of the various header files only need to be known if the user wishes to modify parts of the Cmicro Library.

Functions of the Basic Cmicro Kernel

The list in the following section gives an overview of the functions exported by each module of the Cmicro Library in order to understand the module structure. The functions declared as static are not considered here.

If there is a reference to `xmk_RAM_ptr`, this can be replaced with a * (star) usually. This means that for example the declaration

```
xmk_T_CMD_QUERY_QUEUE_CNF xmk_RAM_ptr qinfo
```

can be replaced with

```
xmk_T_CMD_QUERY_QUEUE_CNF *qinfo
```

which means to refer to a pointer to an object of the type `xmk_T_CMD_QUERY_QUEUE_CNF`.

Exported from env.c

xInitEnv

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is called by the Cmicro Kernel during initialization of the SDL system. The user may include initialization of the environment here.

xInEnv

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is called by the Cmicro Kernel continuously to retrieve signals polled from the environment. Use the Cmicro Kernel function `xmk_Send`* to put signals into the system. The use of this function is not absolutely necessary in the case where the Cmicro Kernel is scaled to preemption and all external Events are put into the SDL system via an Interrupt Service Routines.

xOutEnv

Parameters:

```
In/Out: xmk_T_SIGNAL             sig,
        xmk_T_PRIO               prio,
        unsigned char            data_len,
        void                    *p_data,
        xPID                     Receiver

Return: xmk_OPT_INT
```

This function is called by the Cmicro Kernel if an SDL signal is to be sent to the environment.

Note:

The user has several possibilities to send signals to the environment. Please refer to the subsection about the "Functions of the Expanded Cmicro Kernel" on page 3499.

The function must return with XMK_TRUE, if the Signal was sent to the environment, otherwise it must return with XMK_FALSE.

xCloseEnv

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is called by the Cmicro Kernel during the exit phase of the SDL system. The user may include de-initialization of the environment here.

Exported from mk_user.c

xSDLOpError

Parameters:

```
In/Out: char *xmk_String1 - SDL ADT operators name
        char *xmk_String2 - The reason for failure
Return: -no-
```

This is a function which is to filled up by the user. The function is an central error handling function for ADTs. It is compiled only if XEC-SOP was defined in ml_mcf.h.

ErrorHandler

Parameters:

```
In/Out: int ErrorNo - the given error number
Return: -no-
```

This is a function which is to be filled out by the user. The Cmicro Kernel as well as the SDL application are the main clients of this function.

The user may distinguish between the different errors and define a specific reaction.

The different errors defined in the file `ml_err.h` below the Cmicro Kernel directory.

WatchdogTrigger

Parameters:

```
In/Out: -no-
Return: -no-
```

Description:
If selected this function is called by the Cmicro Kernel each time an SDL transition is executed

> **Caution!**
>
> Be sure, that the time-out used for the Watchdog is longer than the longest SDL Transition (in the case of non preemptive Cmicro Kernel). If the preemptive Cmicro Kernel configuration is used, then the Watchdog Trigger should not be used because the execution time of transitions cannot be calculated.

xRouteSignal

Parameters:

```
In/Out: xmk_T_SIGNAL xmk_TmpSignalID - Signal ID
Return: xPID  Process - PID or xNULLPID
        if no receiver is defined for the given
        signal.
```

Description:
This function is called by the Cmicro Kernel, if SDL signals have no receiver. (undef XMK_USE_RECEIVER_PID_IN_SIGNAL) This might be useful in very small systems in order to spare some RAM memory.The following restrictions apply:

- The SDL System must not contain anything other than the following process declaration. (x,1)
- For each Signal in the SDL System, there is to be only one Receiver process (no signal may be sent to more than one process type).
- no dynamic process creation is used (Create-Symbol is not used)

Note:

Timers are represented as signals, that's because `xRouteSignal` also has to map timers to the receiver process

Hint:

Each signal and timer is represented by a system wide unique integer number.

Exported from mk_main.c

xmk_InitSDL

Parameters:

```
In/Out: -no-
Return: -no-
```

This C function is called by the user before calling the C function `xmk_RunSDL()` and implements the initialization of the whole SDL system, namely Timer, Queue, Processes.

xmk_RunSDL

Parameters:

```
In/Out: -no-
Return: -no-
```

This function operates endlessly unless `XMK_USE_SDL_SYSTEM_STOP` is activated. Then the function is returned if the signal queue is empty or no process is alive. Before processing signals, SDL time-outs are checked and the C function `xInEnv` is called.

xmk_MicroTesterInit

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is used to tell the target configuration to the SDL Target Tester. First the communication interface is initialized, then the startup message is sent to the host. After that the target waits for a "go forever" message from the host.

xmk_MicroTesterDeinit

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is responsible for closing the communication interface to the host after the system has ended.

Exported from mk_sche.c

xmk_StartProcesses

Parameters:

```
In/Out: -no-
Return: -no-
```

This function implements the start-up phase of the SDL system. All static process-instances are created. This means executing the start-transition of all process-instances to be created. For each created process-instance, the first state is set. If configured right, the values SDL_SELF, SDL_PARENT and SDL_OFFSPRING are correctly initialized (only necessary if no semantic check was performed, i.e. if the Analyzer is not used).

xmk_ProcessSignal

Parameters:

```
In/Out: -no-
Return: -no-
```

This function processes an SDL signal and remains in an internal loop, until a signal has been processed or until no signal remains in any input-port in the SDL system.

xmk_CreateProcess

Parameters:

```
In/Out: ProcessID       - ID of the process type
Return: xmk_T_INSTANCE  - created Instance number ID
```

This function tries to create an instance of the given process-type. This can fail, either if the create signal cannot be allocated (no more memory) or if there is no free process instance of that type. E.g. if there is no instance in the `DORMANT` state.

The return value contains the process instance number ID, if one more process instance could be allocated. The return value is set to `xNULLINST` if the creation failed for some reason.

The process instance number is not the same as the process ID. The process ID is calculated from the process ID type plus the process ID instance number.

The generated C code does not use the return value, because the SDL offspring and the parent pid value are stored in the pid tables of the process instance.

xmk_IsAnyProcessAlive

Parameters:

```
In/Out: -no-
Return: xmk_T_BOOL
```

This function checks for any active instance of a process type by searching for instances not in the state XDORMANT.

The function returns with `XMK_TRUE`, if there is an active instance. It returns with `XMK_FALSE` if there is no instance active within the system.

xmk_IfExist

Parameters:

```
In/Out: xPID - Process ID of the process to be
checked
Return: xmk_T_BOOL
```

This function checks if the given PID is valid or not. The return value is `XMK_TRUE` if the given PID is valid. If an instance with this PID does not exist, it returns with `XMK_FALSE`.

xmk_CheckNullPointerValue

Parameters:

```
In/Out: void*
Return: -no-
```

This function checks whether there is a pointer value in the SDL system that is used but has no value. The ErrorHandler is called with the right error message then. The error must be caught in the user's ErrorHandler in order to implement the right reaction on this fatal situation.

xmk_InitPreemptionVars

Parameters:

```
In/Out: -no-
Return: -no-
```

The variables used in preemption are initialized.

xmk_DisablePreemption

Parameters:

```
In/Out: -no-
Return: -no-
```

The variable which stores the preemption status is incremented. A value greater than zero means it is not allowed to perform a context-switch at the moment.

xmk_EnablePreemption

Parameters:

```
In/Out: -no-
Return: -no-
```

The variable which stores the preemption status is decremented if preemption was disabled. If the variable's value is zero after it is decremented, the function `xmk_CheckIfSchedule()` is called.

xmk_FetchHighestPrioLevel

Parameters:

```
In/Out: -no-
Return: xmk_T_PRIOLEVEL
```

This function searches for signals in the priority queue levels. This is done with decreasing priority in order to find the highest priority level at which signals exist. The return value is the highest priority level that contains a signal to process.

xmk_CheckIfSchedule

Parameters:

```
In/Out: -no-
Return: -no-
```

It is checked whether a context switch is admissible. If this is the case the current priority-level is compared with the highest priority-level where a signal exists. Supposing the highest level is higher than the current, a context-switch is performed using `xmk_SwitchPrioLevel()`. This is repeated until the current priority-level is the highest level.

xmk_SwitchPrioLevel

Parameters:

```
In/Out: xmk_T_PRIOLEVEL NewPrioLevel
Return: -no-
```

The global variables for the current priority-level are stored. Afterwards, the function `xmk_ProcessSignal()` is called in order to deal

with the signals on the higher priority level. After returning from this function call the variables for the current priority-level are restored.

With the `NewPrioLevel` the next prio-level to deal with is specified.

xmk_KillProcess

Parameters:

```
In/Out: xPID Process ID
Return: xmk_T_BOOL
```

This function sets a process instance into the state `XDORMANT`, removes all signals directed to it from the queue and resets the local instance data. The return values are `XMK_TRUE` if the call was successfully and `XMK_FALSE` if the process was non-existent, currently running or already in the `XDORMANT` state.

Exported from mk_outp.c

xmk_SendSimple

Parameters:

```
In/Out:
  #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
    xmk_T_SIGNAL  sig
    xPID          Receiver
  #else
   xmk_T_SIGNAL   sig
  #endif

Return: -no-
```

This is a simple SDL output function which needs a maximum of only 2 Parameters. Most SDL systems consist of a lot of “normal” Signals without any parameters and no priority. It makes sense to use this simple function whenever possible to spare program code. The signal is put into the linked list of signals by using a default priority.

With the first parameter `sig`, the signal ID of the signal that is to be sent is specified. With the (optional) second parameter, the receiver process ID is specified. If XMK_USE_RECEIVER_PID_IN_SIGNAL is not defined, the user must implement the C function `xRouteSignal()` which is responsible to derive the receiver from the signal ID in that case. Using `xRouteSignal()` is recommended only if the last few bytes must be spared for transferring of signals.

xmk_Send

Parameters:

```
In/Out:
        xmk_T_SIGNAL sig

        #ifdef XMK_USE_SIGNAL_PRIORITIES
          xmk_T_PRIO prio
        #endif

        #ifdef XMK_USED_SIGNAL_WITH_PARAMS
          xmk_T_MESS_LENGTH data_len
          void xmk_RAM_ptr  p_data
        #endif

        #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
          xPID Receiver
        #endif

Return: -no-
```

This is, compared to the `xmk_SendSimple` function, a complete SDL output function which needs all possible signal parameters.

With the parameter `sig`, the signal ID is to be specified.

With the `prio` parameter, the signal's priority is to be specified (if conditionally compiled).

With the `data_len`, the number of bytes as the signal's parameters is to be specified. The number of bytes is evaluated by using a `sizeof (C struct)` construct. If the signal carries no parameters, this value must be set to 0 (if conditionally compiled).

With the `p_data` parameter, a pointer to the memory area containing the parameter bytes of the signal is given. The memory area is not treated as dynamically allocated within this function. Because the function copies the parameter bytes, the caller may use any temporary memory (for example memory allocated from the C stack by declaring a C variable). This parameter should be set to `NULL` if no parameter bytes are to be transferred (if conditionally compiled).

With the last parameter `Receiver`, the PID of the receiving process is to be specified (if conditionally compiled).

If the XMK_USE_SIGNAL_PRIORITIES is not defined, the signal priorities which are specified with #PRIO in the diagrams is just ignored. The use of signal priorities is not recommended because the violation of SDL. A few bytes can be spared if signal priority is not used.

The XMK_USED_SIGNAL_WITH_PARAMS is automatically generated into the `sdl_cfg.h` file, from the Cmicro SDL to C Compiler. For tiny systems, if there are no SDL signals with parameters specified, this is undefined. It will reduce the amount of information which is to be transferred for each signal with a few bytes.

If XMK_USE_RECEIVER_PID_IN_SIGNAL is not defined, the user must implement the C function `xRouteSignal()` which is responsible to derive the receiver from the signal ID in that case. Using `xRouteSignal()` is recommended only if the last few bytes must be spared for transferring of signals.

XMK_SEND_ENV

Parameters:

```
In/Out: xPID Env_ID

        xmk_T_SIGNAL sig

        #ifdef XMK_USE_SIGNAL_PRIORITIES
          xmk_T_PRIO prio
        #endif

        #ifdef XMK_USED_SIGNAL_WITH_PARAMS
          xmk_T_MESS_LENGTH data_len,
          void xmk_RAM_ptr  p_data
        #endif

        #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
          xPID Receiver
        #endif

Return: -no-
```

This function is to be called when a signal is to be sent into the SDL system, e.g. within the users `xOutEnv()` function. It must be called with one more parameter than the `xmk_Send()` function, which is the first parameter `Env_ID`.

The macro internally uses some variables which are to be declared before the macro can be used. For example in the xOutEnv function the `XMK_SEND_TMP_VARS` macro must be introduced for declaring these variables.

The function is implemented as a macro in C.

xmk_Determine_Receiver

Parameters:

```
In/Out: xmk_T_PROCESS proc_type - process type ID
```

```
Return: xPID                        - process PID
```

This function is used in the context of SDL output in generated C code. The function determines if any instance with the given process type-ID is available to receive the signal. If no instance can be found, `xNULLPID` is returned.

Exported from mk_queu.c

xmk_InitQueue

Parameters:

```
In/Out: -no-
Return: -no-
```

This function initializes the signal queue. It must be called before any other Cmicro Kernel function, e.g. before `xmk_InitSDL()`. All relevant pointers are initialized. All signal-elements are put into the free-list. The SAVE-state of all signals is set to false.

xmk_FirstSignal

Parameters:

```
In/Out: -no-
Return: xmk_T_MESSAGE*
```

The first signal in the current queue which is the one with the highest priority, is copied to the pointer of the currently treated signal and returned to the caller.

The function returns a pointer to the first signal in the queue or `NULL`, if there are no signals in the queue.

xmk_NextSignal

Parameters:

```
In/Out: -no-
Return: xmk_T_MESSAGE*
```

The signal following the current signal is copied to the current signal. If there are no more signals, NULL is returned.

xmk_InsertSignal

Parameters:

```
In/Out: xmk_T_MESSAGE xmk_RAM_ptr p_Message
Return: -no-
```

This function puts a signal into the queue. The position depends on only the signal priority, if specified.

With the parameter `p_Message` a pointer to the signal which is to be inserted is given.

xmk_RemoveCurrentSignal

Parameters:

```
In/Out: -no-
Return: -no-
```

The signal which was currently processed is removed from the queue and inserted into the list of free signals.

xmk_RemoveSignalBySignalID

Parameters:

```
In/Out: xmk_T_SIGNAL SignalId
Return: -no-
```

This function is not used if timers with parameters are in the system (`XMK_USED_TIMER_WITH_PARAMS` macro is defined).

Signals of a given signal-code sent to the current process are removed from the queue and inserted in the list of free signals. The signal currently being processed must not be removed, as it is necessary for the current actions. It is only removed after processing is complete.

With the parameter `SignalID`, the signal ID of the signals which are to be removed for the currently active process instance is specified.

xmk_RemoveTimerWithParameter

Parameters:

```
In/Out: xmk_T_SIGNAL SignalId
In    : SDL_Integer  TimerParValue
Return: -no-
```

This function is only compiled if timers with parameters are in the system (`XMK_USED_TIMER_WITH_PARAMS` macro is defined).

This function has the same purpose as the above `xmk_RemoveSignalBySignalID` but, in addition, must look to the timer's parameter.

xmk_IsTimerInQueue

Parameters:

```
In/Out: xmk_T_SIGNAL TimerID
#ifdef XMK_USED_TIMER_WITH_PARAMS
In    : SDL_Integer  TimerParValue
#endif
Return: -no-
```

This function checks if the given timer is in the signal queue. If timers with parameters are used, the `TimerParValue`, which is conditionally compiled, is checked also.

xmk_RemoveSignalsByProcessID

Parameters:

```
In/Out: xPID ProcessID - PID of Process
Return: XMK_TRUE   - Signal removed
        XMK_FALSE  - no Signal removed
```

All signals addressed to a specific process are removed by calling this Function.

With the parameter `ProcessId`, the PID of the process for which all the signals are to be removed from the queue is specified.

xmk_AllocSignal

Parameters:

```
In/Out: -no-
Return: xmk_T_MESSAGE xmk_RAM_ptr
```

According to the principle chosen by the user (the user has to choose between `XMK_USE_STATIC_QUEUE_ONLY` and `XMK_USE_STATIC_AND_DYNAMIC_QUEUE`), and if possible, an initialized signal instance is returned. The signal instance is then either taken from the static memory pool or from the dynamic memory pool.

Allocation from the dynamic memory pool takes place by calling the `xAlloc` C function.

If `XMK_USE_STATIC_QUEUE_ONLY` is set, the `ErrorHandler` is called with the error "`ERR_N_NO_FREE_SIGNAL`". This indicates that no more memory is available to create one more signal instance, and the user may react appropriately in the `ErrorHandler` C function.

Otherwise, if `XMK_USE_STATIC_AND_DYNAMIC_QUEUE` is set, and if it is impossible to allocate one more instance from the predefined static memory pool, the `ErrorHandler` is called with the error "`ERR_N_NO_FREE_SIGNAL`". This indicates that dynamic memory allocation is started now, and the user may react appropriately in the `ErrorHandler` C function (for example, he might want to print out a warning message). If it is impossible to allocate one more signal from the dynamic memory pool, the `ErrorHandler` is called with the error "`ERR_N_NO_FREE_SIGNAL_DYN`". The user may then also decide what to do in the `ErrorHandler` C function.

A pointer to the signal that was allocated is returned usually, or `NULL`, if there is no space left to allocate one more signal.

xmk_FreeSignal

Parameters:

```
In/Out: xmk_T_MESSAGE xmk_RAM_ptr p_Message
Return: -no-
```

The given signal is de-allocated again.

If the signal was allocated from the static memory pool, it is returned to that pool by initializing it and inserting it into the free list at the first position.

Otherwise, if the signal was allocated from the dynamic memory pool, it will be returned to that pool (by calling the `xFree` C function).

The parameter `p_Message` must point to the signal that is to be initialized.

xmk_TestAndSetSaveState

Parameters:

```
In/Out: xmk_T_STATE State
Return: xmk_T_BOOL
```

This function checks whether a signal's SAVE-state is set or not. In testing, the SAVE-state is set to `TRUE` or `FALSE`.

With the parameter `State`, the value of the process' current state is to be specified.

The function returns with `XMK_TRUE`, if the given state equals the signal's SAVE-state, but returns with `XMK_FALSE` if the given state differs from the signal's save state.

xmk_QueueEmpty

Parameters:

```
In/Out: -no-
Return: -no-
```

This function tests whether there is at least one signal remaining in the queue(s) or not.

The function returns `XMK_TRUE`, if there are no signals in the queue, but returns `XMK_FALSE` if there is at least one signal in the queue. It does not matter if the signal is a saved signal or not.

When the Cmicro Kernel is configured for preemption, all the queues of the different priority levels are checked.

Exported from mk_tim1.c

xmk_InitTimer

Parameters:

```
In/Out: -no-
Return: -no-
```

All initialization of timers is performed within this function, which is called during SDL system start. It initializes some pointers and the free list of timers.

xmk_TimerSet

Parameters:

```
In/Out: xmk_T_TIME   time
        xmk_T_SIGNAL sid
#ifdef XMK_USED_TIMER_WITH_PARAMS
In    : SDL_Integer  TimerParValue
#endif
Return: -no-
```

This function activates an instance of a timer with the given “signal ID” value and the given time. If timers with parameters are used, the `TimerParValue`, which is conditionally compiled, is used also (the value is set to 0 outside this function call if it is not a timer with parameter).

Working principles:

- If a timer instance of this type is already running, this will be deactivated i.e. reset and then set.
- If no free timer is available, the `ErrorHandler()` is called.
- If all is satisfactory, a timer instance is created for the currently running process.

The first parameter `time` specifies the time at which the timer should expire. The call to SDL now is performed in generated C code.

The second parameter `sid` specifies the ID of the timer that is to be started.

If a timer instance of this type is already running, this will be deactivated. If no free timer is available, the ErrorHandler is called. After all these checks a timer instance is created for the currently running process.

xmk_TimerReset

Parameters:

```
In/Out: xmk_T_SIGNAL sid
#ifdef XMK_USED_TIMER_WITH_PARAMS
In    : SDL_Integer  TimerParValue
#endif
Return: -no-
```

This function resets the timer with the given “signal ID” value, if it is active and set by the currently running process. If an active timer instance is found, then the timer is inserted into the free-list. If timers with parameters are used, the `TimerParValue`, which is conditionally compiled, is used also (the value is set to 0 outside this function call if it is not a timer with parameter).

The parameter `sid` specifies the ID of the timer which is to be reset.

xmk_TimerActive

Parameters:

```
In/Out: xmk_T_SIGNAL sid
#ifdef XMK_USED_TIMER_WITH_PARAMS
In    : SDL_Integer  TimerParValue
#endif
Return: xmk_T_BOOL
```

This function checks if a timer with the given ’Signal-ID’ value is active in the current running process. If timers with parameters are used, the TimerParValue, which is conditionally compiled, is used also (the value is set to 0 outside this function call if it is not a timer with parameter).

The parameter `sid` specifies the ID of the timer which is to be checked if it is active.

The function returns with `XMK_TRUE` if the timer is active in the process that is currently running, it returns `XMK_FALSE` otherwise.

xmk_ChkTimer

Parameters:

```
In/Out: -no-
Return: -no-
```

The main intention of this function is to check, if any of the timers which are in the list of active timers is expired. There is a list of active timers in which the active timers are stored in the order of their expiry.

This means that the timer that expires first stands in front of the list.

If a timer was recognized as expired, a timer signal is sent to the owning process instance, the timer is removed from the list of active timers and restored into the list of free timers.

If there is no timer that can expire, the function returns immediately.

Another task for this function is to check if the global system time is overrun. If this case occurs, the `xmk_ChkTimer()` function will readjust the global system time (by calling `xmk_SetTime()` from `mk_cpu.c`) and will readjust all the timers in the active list.

xmk_ResetAllTimer

Parameters:

```
In/Out: xPID pid
Return: -no-
```

All active timers of the given process instance are reset. This will occur if a process instance stops.

With the `pid` parameter the process instance is addressed.

Exported from mk_stim.c

xmk_InitSystime

Parameters:

```
In/Out: -no-
Return: -no-
```

This function initializes the hardware timer and is to be filled out by the user.

xmk_DeinitSystime

Parameters:

```
In/Out: -no-
Return: -no-
```

This function de-initializes the hardware Timer and is to be filled out by the user.

xmk_SetTime

Parameters:

```
In/Out: xmk_T_TIME time
Return: -no-
```

This function sets the system time to the given value `time` and is to be filled out by the user.

xmk_NOW

Parameters:

```
In/Out: -no-
Return: xmk_T_TIME
```

This function is used by

- the Cmicro Kernel to calculate if a timer has expired and this is only the case if absolute time is used by the selected timer model.
- SDL applications to retrieve the current time

The current SDL system time is returned.

The function is to be filled out by the user.

Exported from ml_mem.c

xmk_MemInit

Parameters:

```
In/Out: char* _mem_begin
        char* _mem_end
Return: -no-
```

This function is to be called by the user, before dynamic memory management can be used. The user has to specify the beginning and the end of the area to be used for dynamic memory management. The function should be called before the first call to `xAlloc()`, e.g. as the first statement in the user's `main()` function.

xmk_Malloc

Parameters:

```
In/Out: unsigned long rsize
Return: void *
```

This function allocates one block of memory from the dynamic memory pool. It uses a first fit policy.

xmk_Calloc

Parameters:

```
In/Out: unsigned long RequestedSize
Return: void*
```

This function allocates one block of memory from the dynamic memory pool. It is the same as `xmk_Malloc()` but sets the allocated block to zero.

xmk_Free

Parameters:

```
In/Out: void *mem
Return: void
```

This function is the counterpart of `xmk_Malloc()`. A memory block is de-allocated again.

xmk_Memshrink

Parameters:

```
In/Out: void  *pMemBlock
        unsigned long NewSize
Return: -no-
```

This function is an extension of the compared with the standard of dynamic memory management supported by usual C compilers. It allows the user to shrink down a memory area which was previously requested with `xmk_Malloc()`.

memset

Parameters:

```
In/Out: char *p
        char val
        int length-
Return: -no-
```

This function is a template for the `memset()` implementation.

Caution!

Take care when the preemption policy is utilized.

memcpy

Parameters:

```
In/Out: char *dest
        char *source
        int length
Return: -no-
```

This function is a template for the `memcpy()` implementation.

Caution!

Take care when the preemption policy is utilized.

Exported from ml_mon.c

xmk_GetErrorClass

Parameters:

```
In/Out: xmk_T_ERR_NUM EightBitNumber
Return: int
```

This function returns the error class assigned to the given error number given by the caller. The error class can be one of `XMK_FATAL_CLASS` or `XMK_WARNING_CLASS`. This can be used to classify the predefined error and warning messages and to react properly if such a message occurs.

xmk_MonError

Parameters:

```
In/Out: FILE *fp
        int nr
Return: -no-
```

This function is used to evaluate an ASCII error text from an errornumber given by the caller. The first parameter `fp` must contain a pointer to a valid (i.e. file is opened) file descriptor, for example it could be `stdin`, `stdout`, or a file which was opened with `fopen`. The second parameter `nr` contains one of the possible error numbers defined in `ml_err.h`.

xmk_MonPID

Parameters:

```
In/Out: char *ostring
        xPID pid
Return: -no-
```

Used for test purposes, free use.

xmk_MonHexSingle

Parameters:

```
In/Out: char *p_text
        unsigned char *p_adress
        int length
Return: -no-
```

Used for test purposes, free use.

xmk_MonHex

Parameters:

```
In/Out: char *p_text
        unsigned char *p_adress
        int length
Return: -no-
```

Used for test purposes, free use.

xmk_MonHexAsc

Parameters:

```
In/Out: -no-
Return: -no-
```

Function for test purposes, free use.

xmk_MonConfig

Parameters:

```
In/Out: -no-
Return: -no-
```

Function for test purposes, free use.

Exported from mk_cpu.c

xmk_PutString

Parameters:

```
In/Out: char * Param1
Return: -no-
```

A template for how to print out a character string. The character string must be terminated with "`\0`".

The function must return immediately, i.e. may not be blocking on the output device. If the user does not care about this restriction, the correct function of other program parts cannot be guaranteed.

xmk_GetChar

Parameters:

```
In/Out: -no-
Return: int
```

This function checks if the user has pressed a key on the keyboard (unblocked). The function must return immediately, i.e. may not be blocking on the input device. If the user does not care about this restriction, the correct function of other program parts cannot be guaranteed.

xmk_printf

Parameters:

```
In/Out: char* format
        format string
        other parameters
Return: -no-
```

This function is called from any place in the Cmicro Library, Cmicro Kernel or generated C code, if the `printf` functionality is compiled with at least one of the `XMK_ADD_PRINTF*` defines.

The function must return immediately, i.e. may not be blocking on the output device. If the user does not care about this restriction, the correct function of other program parts cannot be guaranteed.

The function can be used for ANSI C compilers only, because optional argument lists are used like in the `printf` function of the standard C library.

The return value has no meaning and is introduced just for compatibility with `printf`.

xAlloc

Parameters:

```
In/Out: xptrint  Size
Return: void *
```

This function is called from any place in the Cmicro Library, Cmicro Kernel, SDL Target Tester or generated C code, if memory is to be allocated dynamically. The user may choose between the dynamic memory allocation functions from the C compiler or operating system or the dynamic memory allocation functions from Cmicro.

The return value points to the allocated buffer or is `NULL` if the operation was unsuccessful.

xFree

Parameters:

```
In/Out: void **
Return: -no-
```

This is the counterpart of `xAlloc()`. The function is called when a memory block that has been allocated with `xAlloc()` can be de-allocated again. The parameter is the address of the pointer to the allocated buffer.

Example 581 Using the `xFree` function

```
unsigned char *ptr;
ptr = xAlloc(100);
xFree (&ptr);       /* NOTE: Not xFree(ptr); */
```

Functions of the Expanded Cmicro Kernel

The expanded Cmicro Kernel offers additional functionality that is usually not necessary for target applications. Extended functionality is for example required by the SDL Target Tester, but may also be required from the user if it comes to target integration.

It is not absolutely necessary to have knowledge of these functions but knowledge of the functions proves useful when it comes to debugging and testing an application.

Functions for Internal Queue Handling

Exported from mk_queu.c

xmk_SaveSignalsOnly

Parameters:

```
In/Out: -no-
Return: xmk_T_BOOL
```

This function tests whether there only are SAVE signals contained in the queue. This can be used in integrations with operating systems.

The function returns `XMK_TRUE`, if there are SAVE signals only in the queue, otherwise it returns `XMK_FALSE`.

Exported from mk_tim1.c

xmk_NextTimerExpiry

Parameters:

```
In/Out: xmk_T_TIME * RemainingTime
Return: xmk_T_BOOL
```

This function looks for the remaining time of the timer that expires next. This is useful for operating system integration. If there is no active timer, the function returns `XMK_FALSE` and the returned parameter `RemainingTime` is set to 0.

If there is an active timer, the function returns with `XMK_TRUE` and the returned parameter `RemainingTime` contains the time at which the timer expires next. `XMK_TRUE` means there is a timer active and the remaining time is returned in the parameter. `XMK_FALSE` means that there is no timer active and returned parameter is set to 0.

Functions to get System Information

Exported from mk_sche.c

xmk_GetProcessState

Parameters:

```
In/Out  xPID pid
Return: xmk_T_STATE
```

The function evaluates the current SDL state of the SDL process `pid` with the given process PID and returns it to the caller. During the SDL system start-up, one possible return value might be `XSTARTUP`. This means that a process will be started during system start-up. During normal execution and system start-up, the return value `XDORMANT` may be returned. This means that an SDL process instance is either stopped or has never been created dynamically (created).

xmk_SetProcessState

Parameters:

```
In/Out: xPID pid,
        xmk_T_STATE state
Return: xmk_OPT_INT
```

The function sets the SDL-state of the process `pid` that is addressed with the first parameter to the given value `state` within the second parameter. Note, that the value is not checked, because Cmicro does not store any information about this in the generated transition tables.

xmk_GetProcessInstanceData

Parameters:

```
In/Out: xPID pid
Return: void *
```

The function returns the address of the process instance data of the given process-PID, or XNOTEXISTENT, if the pid is not existent.

xmk_QueryQueue

Parameters:

```
In/Out: xmk_T_CMD_QUERY_QUEUE_CNF * qinfo
Return: -no-
```

This function evaluates the current SDL-Queue-State and returns some information to the caller:

- information about the size of the Q (maximum amount of entries)

- information about traffic load, measured until now. (Users can directly use this to scale the queue. This information is valuable in the case, where the maximum traffic load is reached during execution.)
- Information about the number of entries that are currently in the queue.
- A pointer to the physical address of the queue.

Exported from mk_tim1.c

xmk_QueryTimer

Parameters:

```
In/Out: xmk_T_CMD_QUERY_TIMER_CNF *tinfo
Return: -no-
```

This function evaluates the current state of the SDL-Timer handling and returns it to the caller. The following information is contained in the C structure that is returned:

- Information about the size of the timer list entries.
- Information about traffic load, measured until now. (Users can directly use this to scale the timer list. This is valuable in the case, when the maximum traffic load is reached during execution)
- The current number of entries in the timer list.
- A pointer to the physical address of the timer list.

xmk_FirstTimer

Parameters:

```
In/Out: -no-
Return: xmk_T_TIMER *
```

The function returns a pointer to the first active timer. If there is no active timer in the queue, `NULL` will be returned.

xmk_NextTimer

Parameters:

```
In/Out: -no-
Return: xmk_T_TIMER *
```

The function returns a pointer to the next active timer in the list of active timers. If there is no or one active timer in the queue, `NULL` will be returned.

Exported from ml_mem.c

xmk_GetOccupiedMem

Parameters:

```
In/Out: -no-
Return: size_t
```

This function returns the currently occupied amount of memory from the pool. The pool size is defined with `XMK_MAX_MALLOC_SIZE`.

xmk_GetFreeMem

Parameters:

```
In/Out: -no-
Return: size_t
```

This function returns the amount of available memory from the pool. This means that it returns the difference between the size of the pool (`XMK_MAX_MALLOC_SIZE`) and the currently occupied memory.

xmk_CleanPool

Parameters:

```
In/Out: -no-
Return: int
```

This function reinitialize the memory pool to free memory leaks. The memory pool can only be cleaned if there are no allocated blocks left.

Alternative Function for sending to the Environment

Exported from mk_outp.c

xmk_EnvSend

Parameters:

```
In/Out:
        xmk_T_SIGNAL sig

        #ifdef XMK_USE_SIGNAL_PRIORITIES
          xmk_T_PRIO prio
        #endif

        #ifdef XMK_USED_SIGNAL_WITH_PARAMS
          xmk_T_MESS_LENGTH data_len
          void xmk_RAM_ptr  p_data
        #endif

        #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
```

```
            xPID Receiver
          #endif

   Return: -no-
```

This function is an alternative to the standard function XMK_SEND_ENV for putting signals into the SDL system in external C code. The function is useful when signals are to be sent in interrupt service routines directly. The function is shorter than XMK_SEND_ENV, but performs not so many error checks. It is especially not checked, if the receiver process ID exist. Usually there is also no call to functions which produce trace output because this would lead to problems in interrupt service routines. But for some error situations like either

- no free signal available
- no more memory available

the ErrorHandler() C function, which is to be implemented by the user, is called.

With the parameter sig, the signal ID is to be specified.

With the prio parameter, the signal's priority is to be specified (if conditionally compiled).

With the data_len, the number of bytes as the signal's parameters is to be specified. The number of bytes is evaluated by using a sizeof (C struct) construct. If the signal carries no parameters, this value must be set to 0 (if conditionally compiled).

With the p_data parameter, a pointer to the memory area containing the parameter bytes of the signal is given. The memory area is not treated as dynamically allocated within this function. Because the function copies the parameter bytes, the caller may use any temporary memory (for example memory allocated from the C stack by declaring a C variable). This parameter should be set to NULL if no parameter bytes are to be transferred (if conditionally compiled).

With the last parameter Receiver, the PID of the receiving process is to be specified (if conditionally compiled).

If XMK_USE_SIGNAL_PRIORITIES is not defined, the signal priorities which are specified with #PRIO in the diagrams is just ignored. The use of signal priorities is not recommended because the violation of SDL. A few bytes can be spared if signal priority is not used.

The XMK_USED_SIGNAL_WITH_PARAMS is automatically generated into the `sdl_cfg.h` file, from the Cmicro SDL to C Compiler. For tiny systems, if there are no SDL signals with parameters specified, this is undefined. It will reduce the amount of information which is to be transferred for each signal with a few bytes.

If XMK_USE_RECEIVER_PID_IN_SIGNAL is not defined, the user must implement the C function `xRouteSignal()` which is responsible to derive the receiver from the signal ID in that case. Using `xRouteSignal()` is recommended only if the last few bytes must be spared for transferring of signals.

Technical Details for Memory Estimations

Allocating Dynamic Memory

Introduction

This section shows when and how dynamic memory allocation is used in the Cmicro Package. It shows,

- how dynamic memory is allocated
- how to estimate the memory requirements for an application

The Cmicro Package uses a form of dynamic memory management for the following objects:

- processes
- signals
- timer
- some of the predefined sorts like charstring, ASN.1 sorts and generators

However, real dynamic memory management is used only in one case, namely for SDL signals, if a signal carries parameters with more than a few bytes.

This means that the Cmicro Kernel has its own memory management to handle processes, signals, and timers. This is done in such a way that each of these 3 objects are managed separately. For each of these 3 objects, a separate fixed memory area is reserved during compilation time, i.e. the area that handles processes cannot be reused to handle timers. This seems to be a restriction but in many micro controller applications users have to fix an upper limit of processes, signals and timers which can be handled in parallel during run time.

Processes

Processes are handled without dynamic memory allocation functions. The user has to specify an upper limit of process instances in the SDL diagram separately for each SDL process. For each process instance, there is a variable which is statically allocated.

Signals with and without Parameters

Cmicro signals are both ordinary SDL signals as well as timer signals. Where timers are implemented without parameters and can therefore be regarded as signals without parameters.

- No `malloc`/`free` for timers
- No `malloc`/`free` for signals without parameters
- No `malloc`/`free` for signals with parameters if less than or equal to `XMK_MSG_BORDER_LEN` bytes parameters are to be transferred.
- malloc and free are used for signals with parameters if more than `XMK_MSG_BORDER_LEN` bytes parameters are to be transferred.
- `XMK_MSG_BORDER_LEN` bytes is a macro defined in the manual configuration file `ml_mcf.h`. It can be set to any value, i.e. zero or the maximum number of signal parameters to be handled in the system.

Timers

For timers, no malloc and free functions are used. The Cmicro SDL to C Compiler evaluates the amount of timers in the system and generates a C constant, which is then used in the Cmicro Kernel to define an array for timers.