Hard Real-Time and Synchronous Programming with SDL

Emil Naef

Department of Automatic Control **Lund Institute of Technology** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER THESIS
	Date of issue April 2001
	Document Number ISRN LUTFD2/TFRT—5671--SE

Author(s) Emil Naef	*Supervisor* Karl-Erik Årzén LTH Linus Helmestam TeleLogic
	Sponsoring organization

Title and subtitle
Hard Real-Time and Synchronous Programming with SDL
(Hård realtid och synkron programmering med SDL)

Abstract

This is a report of how the two Telelogic development tools the SCADE Suite, and the SDL Suite can be used together, combining a time-driven language and an event-driven language. Suggestions on how the tools can be integrated are presented. The report also suggests how Telelogic can improve the SDL Suite from a hard real-time aspect. Last part of the report shows how the scheduling algorithm "Earliest Deadline First" can be implemented in the SDL Cmicro kernel, and how the implementation can be improved.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

Contents

1 Introduction

Computer systems are growing larger and more complex today. Where yesterdays systems were small, and often consisting of only one process, executing on one computer, today's systems often contain many interacting processes and can be distributed over many computers. With the growing systems, the demand for developments tools is increasing. Telelogic has developed a tool for development of communication system based on a language called Specification and Description Language, or SDL. SDL is widely used in development of telecommunication and data communication applications. When Telelogic bought Verilog in 1999, the development tool SCADE was acquired. SCADE is a synchronous control system development tool, and since the demand on more complex control systems, where many processes must interact, is increasing, Telelogic wanted to investigate how SCADE and SDL can be used together.

This report contains a short presentation of SCADE and SDL. It presents a solution on how SCADE and SDL can be used together, where time-driven and event-driven program parts are interacting. The report also contains suggestions on how the SCADE and the SDL tools can be integrated to support the user using both tools together.

The last part of the report describes how the scheduling algorithm "Earliest Deadline First" has been implemented in the SDL Cmicro kernel. New scheduling algorithms is a necessary step in order to support hard real-time with the SDL tools.

1.1 Background

Telelogic develops and sells tools for software development. Two of these tools are the SDL Suite and the SCADE Suite. SDL is used for development of communication systems, and is an event-driven and asynchronous language. SCADE is used in development of control system, and is a time-driven and asynchronous language. Combining these two tools, using the advantages of both event-driven and time-driven techniques, can improve Telelogics position as a real-time development tool vendor. Telelogic also wanted to investigate how the SDL Suite can be improved in the hard real-time aspect.

1.2 Purpose

This report will discuss and hopefully answer the following questions:
- How can the SCADE-tool and the SDL-tool be used together?
- How can the SDL tool be improved in the real-time aspect?

The report also shows how a scheduling algorithm can be included in the SDL-kernel.

1.3 Methods

The material in the reference list has been used as background material for the work. For the implementation of the new scheduling algorithm in the Cmicro kernel, a considerable time has been spent on studying the kernel C-code. Apart from the studies, asking Telelogic employees has helped a lot in the creation of this report.

1.4 Structure of This Paper

Chapters 2-4 are an introduction to the SCADE tool and the SDL tool. Chapter 5 shows how SDL and SCADE can be used together. Chapter 6 presents suggestions on how to improve the SDL tool to work better with SCADE. Chapter 7 describes how a scheduling algorithm has been introduced to the Cmicro kernel.

2 SCADE

SCADE is the abbreviation for Safety Critical Applications Development Environment. SCADE is a development tool from the French company Verilog. The tool is used to develop hard real-time applications. Typical SCADE areas are nuclear control systems and flight control systems.

The SCADE suite is a graphic development tool based on the language SCADE. SCADE is an extended version of Lustre and is compiled to Lustre code and then to C or ADA.

This chapter will start with describing attributes for the synchronous language group, of which Lustre and SCADE are a part. The data-flow model on which SCADE and Lustre are based will also be discussed. The last two sections in this chapter will explain Lustre and some of the SCADE extensions to Lustre. The information in this chapter has been retrieved from [Hal93] except for Section 2.4, where the information regarding SCADE was found in [SCADE].

2.1 Synchronous Languages

Languages like Argos, Esterel and Lustre are called synchronous languages. A language is called synchronous if the output is synchronized with the input. In other words, the program will react instantly on an event and the output will come instantaneously after the input. There is no time between input and output.

Characteristic for a synchronous language is that it uses a multiform notion of time. This means that physical time will be handled as an external event and all events can be used as a clock trigger. The only important time aspects are that two events can happen simultaneously and that the order among events is the same at all time.

Synchronous programs are used in reactive systems. Reactive systems react on input from the environment where the input speed is determined by the environment. A reactive system can never raise an event if the environment does not invoke it. The system is static between events.

The executable code generated by a synchronous language is always sequential, which prevents sharing problems. Only one task is executed at a time, and that task will be executed until it is finished.

Synchronous languages are deterministic, which means that the output is determined only by the input and the program state. It is easier to design, analyze and debug a language if it is deterministic.

A synchronous language is not a complete language; it needs a host language, which the code is compiled into. The host language can provide complex data structures, handle the interface with environment, databases etc.

2.2 Data-flow Model

One approach to construct synchronous programs is the dataflow model. The program is built from interconnected operators that work in parallel to each other. As soon as an operator receives an input, it calculates the output. The dataflow model was and is used mainly in the control and the electronics areas.

The dataflow model is a functional model and the mathematical clarity makes the approach efficient for the use of formal methods for analysis, design and verification. Figure 2.1 shows a dataflow graph and the corresponding equation.

An operator can be created by a set of connected operators. It is therefore easy to create a hierarchical structure, with the advantages of reuse.
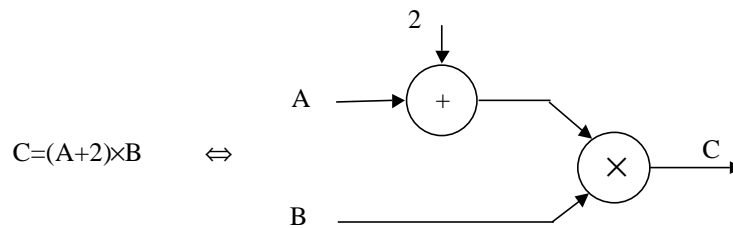
Figure 2.1: A dataflow graph of the equation C=(A+2)*B.

2.3 The Lustre Language

Lustre is a formal synchronous data-flow language, which was designed 1984 by IMAG Institute in Grenoble. Merlin-Gerin Company developed a graphical development tool for Lustre.

Lustre is based on the notion of flows and clocks. A flow is a sequence of values corresponding to a sequence of clock instants. The basic cycle is the fastest cycle in the program from which other slower cycles are derived. Boolean-valued flows are used to define clocks. Each flow, or sequence of values, is mapped to one clock cycle.

Basic Cycle	1	2	3	4	5
Values of C	True	False	True	True	False
Cycles on C	1		2	3	
Values of D	True		False	False	

Table 2.1: Boolean flows and clocks

Table 2.1 shows a flow C on the basic cycle. A new clock cycle can be derived from C, here shown in the row Cycles on C. The D flow is based on the clock cycle derived from C.

Lustre has only three types of variables, booleans, integers, and reals. If other data types are needed, they have to be imported from the host language and handled as abstract data types. Arrays can be defined in Lustre by a tuple constructor.

A variable is declared as a data type and defined by one, and only one, equation. The variable can be substituted by the equation and vice versa everywhere in the program.

Lustre has predefined operators of different types. The operators are arithmetic, Boolean, conditional, relational, or sequential:

- Arithmetic operators are '*+*', '*-*', '***', '*/*', '*div*', and '*mod*'.
- Boolean operators are '*and*', '*or*', and '*not*'.
- Conditional operator is *if* <...> *then* <...> *else* <...>
- Relational operators are '*=*', '*<*', '*<=*', '*>*', '*>=*', and '*<>*'
- Sequence operators, also called temporal operators, are '*pre()*', '*->*', '*when*', and '*current()*'

These operators can be used to create new and more complex operators. The first four groups of operators operate on operands on the same clock. The sequence operators operate on flows.

The *'pre()'* operator remembers the value of the argument from the preceding clock cycle. The operator will return nil at the first clock cycle. If the sequence *E (e1, e2, e3...)* are used as argument to the *'pre()'*-operator, *pre(E)*, the resulting sequence will be *(nil, e1, e2, e3...)*.

The operator '*->*' is used to give flows initial values. For example, the sequences *E (e1, e2, e3...)* and *F (f1, f2, f3...)* are used to create a new sequence *G=E->F*. The resulting sequence will be *(e1, f2, f3...)*. This operator can be used together with the *'pre()'*-operator since the first value in the *'pre()'*-sequence is undefined *(nil)*. Using the sequences in the previous example the expression *G=E->pre(F)* will have the following sequence of values: *G=(e1, f1, f2, f3...)*.

The operator *'when'* is used to change clock frequency. The *'when'*-operation is also called *filter*. If we use sequences *E* and *F* from the *pre*-examples to generate a new sequence *G=E when F*, the *G* sequence will have the same value as *E* when, and only when, *F* is true. When *F* is false the *G* sequence will have no value at all. Note that *F* must be a boolean flow. Table 2.2 shows an example with the *'when'* operator.

E	E1	E2	E3	E4	E5
F		False	True	False	False	True
G=E when F		E2			E5
Current(G)	Nil	E2	E2	E2	E5

Table 2.2: Filtering and projection.

The last sequential operator is the projection operator *'current()'*. It is used to get values from a variable on a slower clock. By using *'current()'* the last value of the argument expression will be projected to the clock instant of the expression which the *'current()'* operator is currently in. Table 2.2 shows an example with projection.

Assertions were initially introduced to help the compiler to optimize better. An assertion, or *'assert()'* as it is called in the language, is used to show that an expression is always true. For example *assert(X=Y)* shows that *X* and *Y* are on the same clock and that they always have the same values. *'assert()'* is also used in program verification.

A program in Lustre is a net of operators. It is often convenient to divide this net into subnets, which can be presented as new operators. A subnet presented as a new operator is called a node. The nodes can be used in any expression in the program. Input and output parameters are defined when the node is declared. A node can have one or more input parameters and one or more outputs. The following example is a counter, with an initial value, an incremental value and a reset event as input parameters and the counter value as output:

```
node COUNTER(init_value, incr_value: int; reset: bool)
returns(N: int);
let
    N = init_value -> if reset then init_value
                            Else incr_value + pre(N);
tel
```

The counter can be called in an expression somewhere else in the program, for example:

```
Number_of_ticks = COUNTER(1, 1, false);
```

Lustre compilers cannot handle cyclic definitions where a variable depend on itself, like *X=3*X+1*. Another constraint is false deadlocks. The Lustre compiler will not accept following code:

```
X = if C then Y else Z
Y = if C then Z else X
```

The reason is the problem of deciding if it is a deadlock or not.

2.4 SCADE Extensions to Lustre

As mentioned before SCADE is an extension to Lustre. New data types and structures are introduced and more predefined operators are available.

SCADE introduces the predefined types character, called *'char'*, and string, *'string'*. These types can only be used in a very limited way. The main reason for the introduction of the types was probably to make it possible to send text messages from SCADE, and not for text handling.

In addition to the textual types there are enumerated, structured, generic and deferred types in SCADE. These types can be used together with some of the predefined operators.

The main difference between Lustre and SCADE is that SCADE is a graphical language while SDL is a textual language. Figure 2.2 below shows the counter in the example in the Section 2.3.2, defined as a SCADE node.
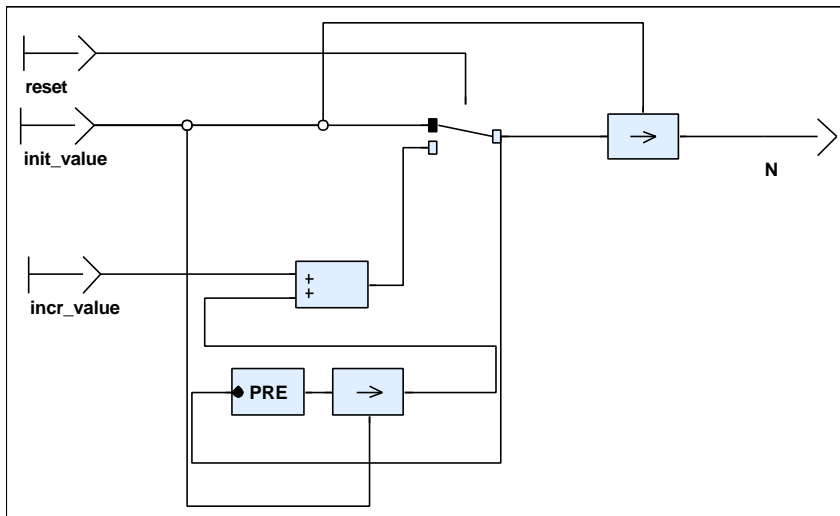
Figure 2.2: SCADE defined Counter

3 SDL

Specification and Description Language, or SDL, is an event-driven language mainly used in telecommunication development. SDL is a formal language, standardized by ITU.

This chapter will explain SDL-96 used in the SDL Suite. The new version SDL-2000 has a few differences to SDL-96, but the two versions work mainly the same way. In the hard real-time aspect there is no difference.

3.1 Overview

In SDL a system is divided into processes, which communicates with each other and with the system environment by signals. To make a system more structured SDL uses blocks to divide the system into subsystems. A block consists of blocks or processes. For an overview see Figure 3.1.

A process in SDL is an extended finite-state machine. This means that the process state is defined by both explicit state variables and process variables. A process is static as long as no handled signal arrives at the process. A handled signal starts a transition, which ends in a new or the same state. The process can send signals to other processes, set timers, calculate variable values, etc. during a transition.

Process priority is not defined in SDL. All processes have the same priority and if more than one process wants to execute FIFO selection are used.
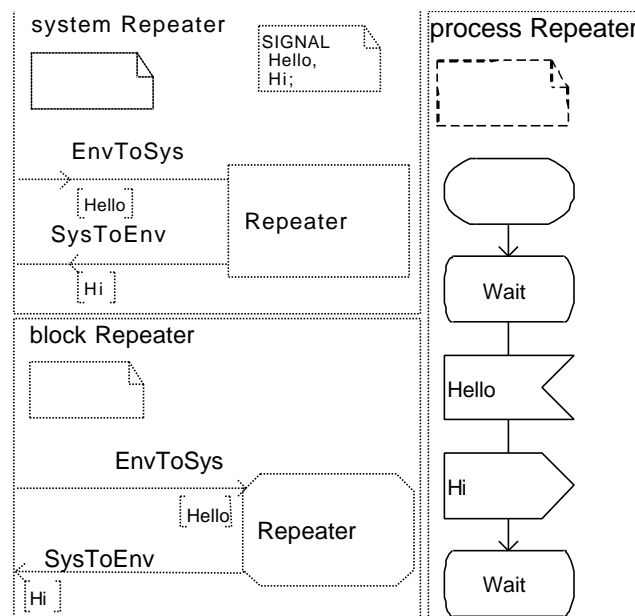
Figure 3.1: System, Block and Process in SDL

A process behavior is defined by symbols in SDL. Figure 3.2 shows the most common symbols used in SDL.
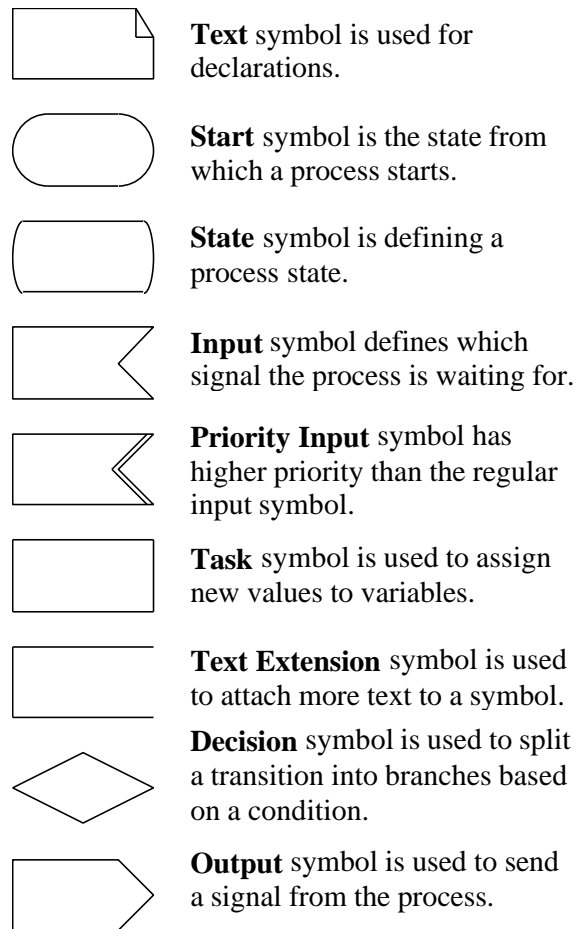
Figure 3.2: SDL Process Symbols

Signals are used in the SDL system for communication between processes and the system environment. The signals follow predefined signal routes between two processes or between a process and the block environment. A connection between blocks or between blocks and system environment is called channel. The difference between a cannel and a signal route is that signal routes do not have any delays, while channels can be either delaying or non-delaying. Figure 3.3 shows Channels and Signal Routes in SDL.

A process may prioritize a signal by using priority input. The prioritized input will be consumed before non-prioritized input.
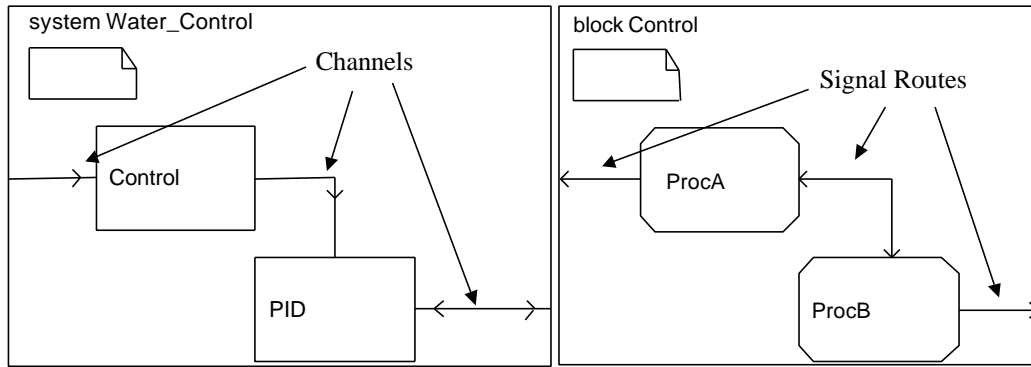
Figure 3.3: Channels and Signal Routes in SDL

Timers are used in SDL to make a transition after a certain time. The *'Set()'* command is used to set the time when the timer shall expire. When the timer expires the process waiting for the timer will receive a signal with the same name as the timer. The *'Reset'* command is used to reset the timer. Figure 3.4 is an example of how timers can be used. Note that the *Wait* state in Figure 3.4 is waiting on both *T1* and *Sig1*. Depending on which signal arrives one of the two transitions are executed.
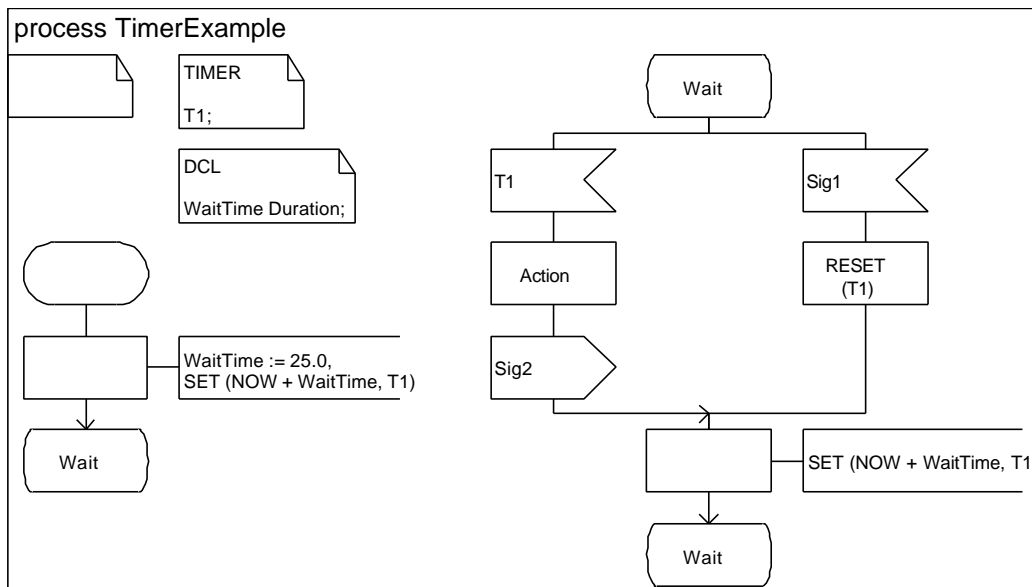
Figure 3.4: Example of an SDL process using a timer.

4 The SDL Suite

The Telelogic Tau SDL suite is the tool Telelogic has developed for specifying, implementing, validating, verifying, and testing SDL systems. The focus in this work is implementation and code generation.

The SDL Suite has three versions of SDL to C code compilers. The compilers are used for different purposes:

- The Cbasic version is used for simulation and validation of the SDL system.
- The Cadvanced version is used for building any type of application.
- The Cmicro version is used to create compact C code, and thereby memory-efficient applications.

Cadvanced and Cmicro will be reviewed in the following sections. We will not focus on Cbasic anymore, since Cbasic cannot be used in hard real-time applications.

4.1 Cadvanced

Cadvanced SDL to C compilation is normally used when creating a SDL system. When generating code from SDL predefined macros are introduced to handle system calls. These macros are defined depending on operating systems and integration model.

In Light Integration, the macros are expanded into standard kernel functions. A Light Integration application will be one thread in the operating system. Signals between two processes in these applications are very time efficient since only pointers to a shared memory are passed between the processes. The Light Integration SDL kernel can be used in environments without real-time operating systems.

Tight Integration uses lower level macros. The Tight Integration uses RTOS primitives to handle the processes as different threads in the RTOS kernel. Signals between processes in Tight Integrated applications are slower than in Light Integrated ones. The reason for this is that a signal message must be copied to the called process memory. Figure 4.1 shows the difference between Light and Tight Integration.
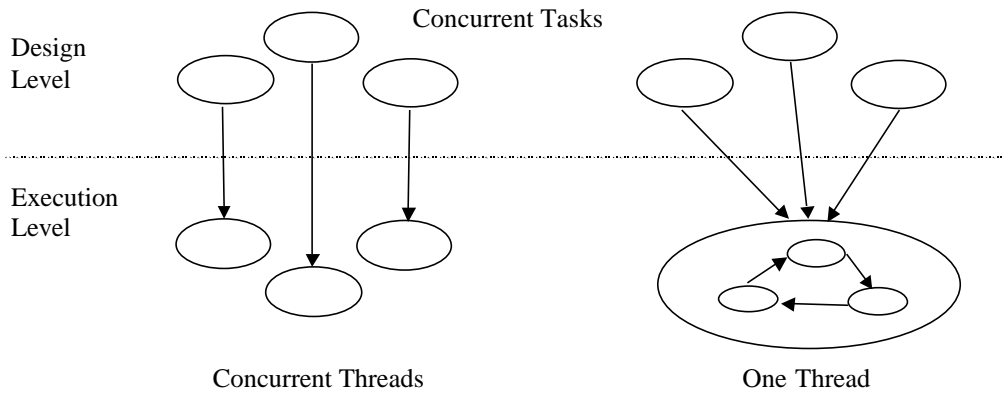
Figure 4.1: Left: Tight Integration, Right: Light Integration

4.1.1 Priority and Preemption

Although SDL does not have any priority, the SDL Suite allows the user to define priority for processes and signals. Process priority is defined from zero and upwards in continuous steps, where zero has the highest priority. All processes with higher priority are processed before a process with a lower priority can be processed. Priority is declared with the process declaration. Figure 4.2 shows how process priority can be declared.

Figure 4.2: Processes priority declaration in SDL

Signal priority is weaker than process priority, which means that a process with higher priority will always execute first regardless of the signal priority. Signal priority levels do not have to be continuous. Signal priority is declared with the signal declaration:

```
Signal
S1 /*#PRIO 12 */,
S2 /*#PRIO 23 */;
```

Preemption can only be used with Tight Integration; Light Integration is always non-preemptive. The operating system must support preemption if an application shall be able to use preemption.

Preemption is disabled during system start up. System start up is the part of the execution where the static processes are created. It is possible to enable and disable preemption during the execution by using the functions

12

xmk_DisablePreemption and *xmk_EnablePreemption*. Disabling the preemption is useful when the program contains critical parts where shared data are accessed.

Only a process with higher priority than the executing can force a context switch when preemption is used. Processes with the same priority cannot interrupt each other.

4.2 Cmicro

Cmicro has a more optimized code than Cadvanced. As a result restrictions on the use of SDL is introduced. Cmicro is typically used in micro controllers, mobile telephones, and so on. Common for Cmicro systems is that they usually work in environment with limited memory and small processor capacity.

Light and Bare Integration are the Cmicro version of Cadvanced Light Integration. Both Light and Bare Integration will create one thread where the SDL program executes. Tight Integration for Cmicro is not a part of the SDL Suite. There are consultants in Telelogic that can do the integration manually.

Bare Integration is used when the target environment does not have any operating system. There is no interface with hardware drivers, hardware functions and interrupt service routines in the Bare Integration. The user has to define these interfaces using macros from the Cmicro library. In Light Integration the operating system provides an interface with hardware, interrupts and so on.

4.2.1 Priority and Preemption

Priority in Cmicro is declared in the same way as in Cadvanced, see Section 4.1. Process priority is only used when the scheduler is preemptive. Preemption is an option in Cmicro, whether to use it or not is depending on the application. If no hard real-time requirements are present the non-preemptive option is recommended. Preemption is only useful in applications with a mix of processes with very different reaction time requirements.

Priority and preemption must be defined with two macros in the *ml_mcf.h* file called Configuration Header File in the Targeting Expert. The macros are defined in the end of the file in the user code section with the lines:

```
#define XMK_USE_PREEMPTIVE
#define MAX_PRIO_LEVELS <number of priority levels>
```


4.3 Timers

As mentioned in Section 3.4, a timer can be set to signal a process after a certain time. If a timer expires the SDL kernel will remove the timer from the timer list and send a signal to the concerned process. The timer signal will be handled as any other signal. If there is a transition when the timer expires, the system will not notice the timer until the transition is finished.

4.4 User Interface Overview

The SDL Suite main window is called the Organizer. It is in the Organizer the user can see an overview of all files in the project. By double-clicking on the different files the corresponding window will pop up. It is also in the Organizer that the user starts programs for different part of the development; one of these is the Targeting Expert. The Targeting Expert is used to choose compile and linking options. It is here the user chooses if the Cadvanced, Cbasic, or Cmicro compiler should be used.

5 SCADE and SDL Together Using Existing Tools

This chapter presents a solution that is possible to use in the SDL Suite today. The solution is based on that SCADE is compiled to C code, and therefore this solution is applicable to all C code.

This paper focuses on one SDL to C code compiler. The chosen compiler is the Cmicro compiler because the Cmicro code is the best fitting code for microprocessors, which is the typical target for a SCADE application.

5.1 Connecting SCADE and SDL

The SDL Suite does not support SCADE, but it does support C code. By using the SDL tool *CPP2SDL* it is possible to translate C code into SDL. After the conversion from C to SDL, the C functions can be called as if they where SDL functions.

An example will be presented to show how SCADE and SDL can be connected. This example calculates the current speed based on distance input. The system calculates the difference between current distance input (*xA*) and last input, and then divides the result with the constant (*dt*). The speed output is stored in *VxA*. The SCADE system in this example can be seen in Figure 5.1.
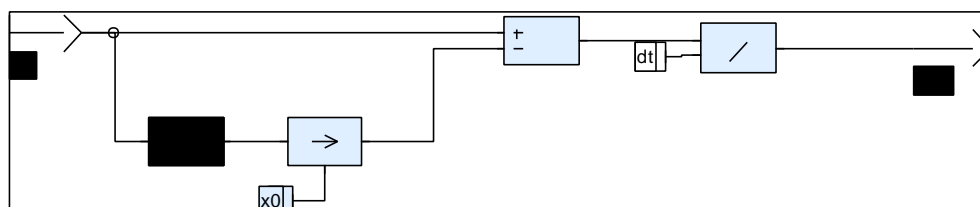
Figure 5.1: SCADE Speed node

The SCADE system is first compiled into C code. There is a lot of code generated, but it is only a part of the *Speed* header file that is interesting in the SDL integration:

```
typedef struct _C_Speed{
_int _I0_xA;
real _O0_VxA;
_int _L5_Speed;
bool _M_init_0_Speed;
} _C_Speed;

void Speed_init (_C_Speed *);
bool Speed (_C_Speed *_C_);
```

The struct contains the input (*_I0_xA*) and the output (*_O0_VxA*) for the system. The *Pre*-value (*_L5_Speed*) is also stored in the struct. The last variable in the struct is set to true by *Speed_init* to initialize the systems values in the *Speed*

function. Note that a pointer to the struct is used as parameter to the two functions.

Speed is the function defined in the SCADE node. This function is not automatically declared in the header file, and must therefore be defined by the user. The *Speed* function can be found in the C-file though. *Speed* uses the input variable in the struct to calculate the new *Pre*-value and the output value, which are returned in the struct. The boolean return value is always set to true.

To be able to use the SCADE generated C code a translation to SDL must be made. The first step is to create a PR-block in the process diagram. A PR-block is a text reference block, which refers to a file. In this case the reference is to an "Import Specification" file. The PR-block can be seen in Figure 5.2, containing the text "PR" and "SCADE". Write a name for the "Import Specification" file used to instruct the compiler how to map the C code functions to SDL directives. By double clicking on the PR-block the Import Specification file will pop up:

```
CPP2SDLOPTIONS {
-generatecpptypes -sdlsorts -c -supportfunctions -dialects ANSI -errorlimit 5 -prefix
ptr=ptr_  arr=arr_  keyword=keyword_  incomplete=incomplete_  tpl=tpl_  -suffix
uscore=uscore
}

TRANSLATE
{
_C_Speed
Speed_init
Speed
}
```

The CPP2SDLOPTIONS-block contains compiler flags used to translate the C code into SDL. The CPP2SDL-Options, found in the Organizer, automatically writes this block. The TRANSLATE part specifies which functions and structs that are going to be mapped to SDL. The user has to define this part of the file.

When the Import Specification file is finished the SCADE system header file must be added to the Import Specification file, in this example *Speed.h*. This is done in the Organizer using "Add new C Header".
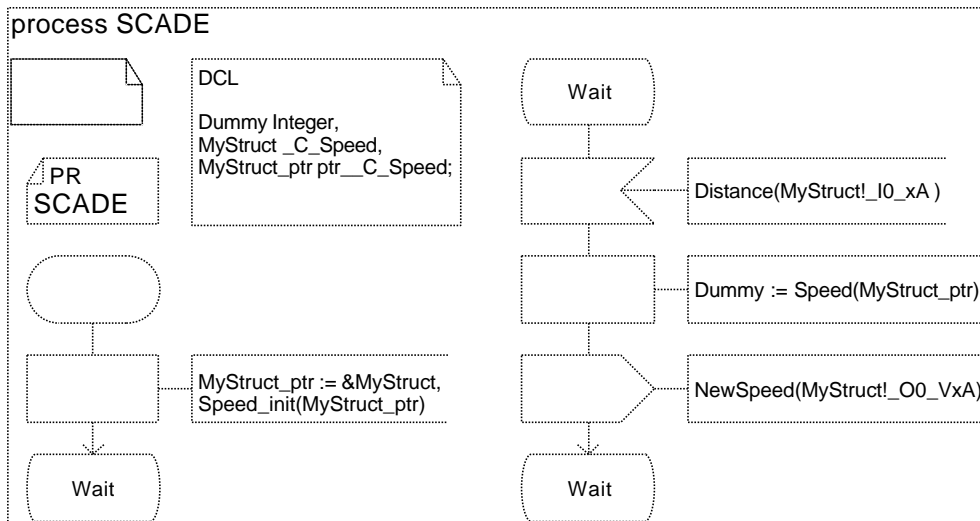
Figure 5.2: SCADE generated C code in a SDL Process

Figure 5.2 shows how the SCADE generated C code can be used in the SDL system. The functions from SCADE are used in the same way as SDL procedures.

Notice that the boolean return from the *Speed* must be handled in SDL. This is done with the *Dummy* variable in this example. The pointers used to pass the struct to the functions are worth an extra notice. First of all the SDL Help recommends not using pointers. The CPP2SDL mapping automatically declare a pointer type when one is needed. In this example a pointer to the struct is needed. The pointer type is available by putting the prefix 'ptr_' before the object or struct name that the pointer is pointing at. In this case *'ptr__C_Speed'* is a pointer type for *'_C_Speed'*. In the SDL diagram in Figure 5.2, *'MyStruct'* and *'MyStruct_ptr'* are the *'_C_Speed'* struct and a pointer to it. The pointer is initialized to point at the struct in the initializing transition with *'MyStruct_ptr := &MyStruct'*.

5.2 Priority, Preemption and Timers

In hard real-time applications priority and preemption are very important attributes to ensure that time constraints are kept.

A process with hard time constraints must have high priority while processes with soft time constraints shall have a low priority. How to define and use priorities is described in Section 4.2.1.

The SDL timer is sufficient if the system only contains processes with equally prioritized transitions, or if the system has soft time constraints. If the system has processes with time-consuming transitions, the time between the defined and the actual timer duration can be considerable, since the expiring timer only can be handled between transitions.

To improve the timer events, an external timer must be used. When the Cmicro kernel is using preemption, external signals will be put into the SDL system with

Interrupt Service Routines. This has the effect that a timer event will be put into the SDL system immediately, and then the concerned process will start a transition if it has the right priority. The use of external timers is of course dependent on if the RTOS (or the hardware) supports timers.

5.3 Response Time

Response time is the time it takes the system to produce an output when a certain input is presented, see Figure 5.3. It is very important to be able to calculate a response time and thereby see if the system meets its time requirements.

$[t_i]$ → System → $[t_o]$
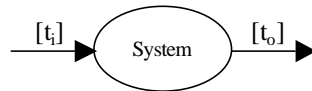
Figure 5.3: Response Time $= t_o - t_i$

The response time for the SDL system can be calculated as:

- The execution time for the called transition if the system is idle (no process running).
- The time for a context switch plus the execution time for the called transition if the system has a process, with lower priority than the called process, running.
- The execution time for all the called processes with higher or equal priority in the system, plus the time for scheduling between the transitions. It is possible to define signal priority and thereby make it possible for a certain transition to be prioritized in a process priority level, see Section 4.1.1.

The SCADE generated code is deterministic, which means that the execution time for a SCADE defined function is constant. It is therefore fairly easy to calculate the execution time for the code. This makes it possible to calculate the response time for a signal to a process with a SCADE defined behavior. The following section will present an explaining example.

5.4 A Time Trigged Example

This section will present an example with a time-trigged system, and show how the response time can be calculated. The system contains one process calling a SCADE node, shown in Figure 5.4, and one process for user communication. The SCADE node is a PI-regulator, used to control a heater.

18
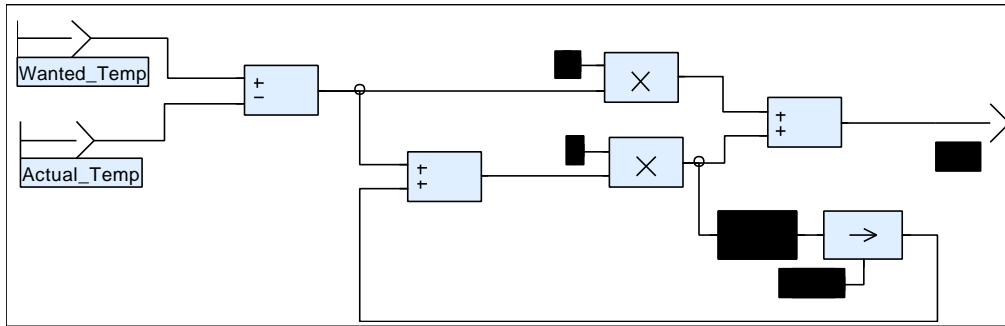
Figure 5.4: SCADE HeatControl node

The SCADE node is compiled and included in the SDL system the same way as the *Speed* node in the previous example in Section 5.1.
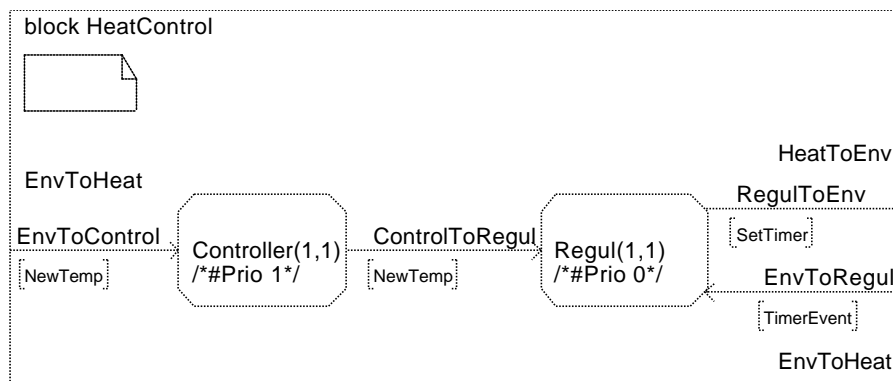
Figure 5.5: The SDL system at block level

Figure 5.5 shows the system at block level. The *Controller* process receives signals, with the desired temperature, from the environment. *Controller* then checks if the desired temperature is inside the heaters working area, and then pass the signals to *Regul*. The *NewTemp* signal is an aperiodic event with soft time constraints. As a result of the soft time constraint, *Controller* has lower priority than *Regul*.

Figure 5.6 shows how *NewTemp* is checked and passed on by *Controller*.
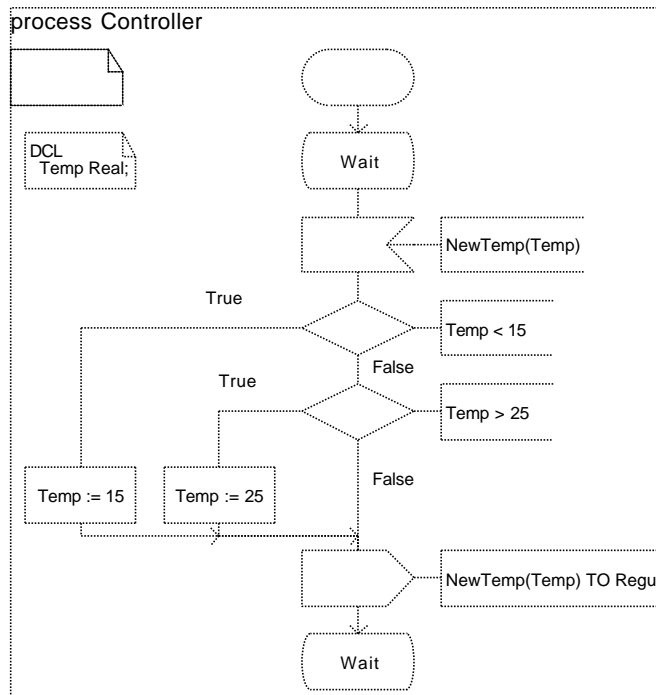
Figure 5.6: Process Controller

Regul is the actual heat controller containing the SCADE node. This process executes periodically by the use of an external timer. The signals *SetTimer* and *TimerEvent* are used to set and receive the external timer.
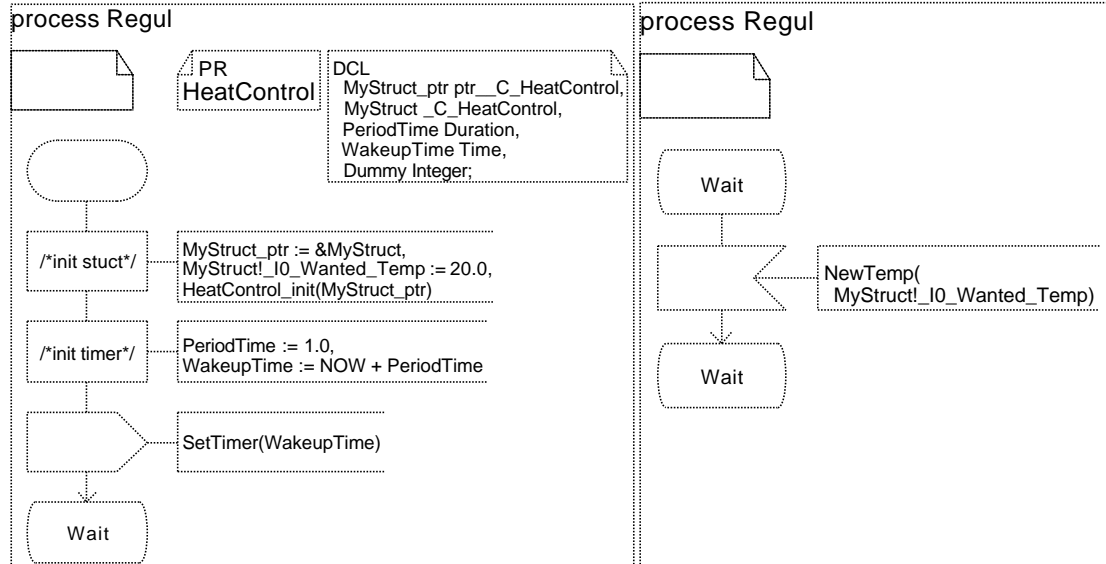
Figure 5.7: Left: Initialization of Regul. Right: New wanted temperature.

Figure 5.7 shows how *HeatControl* and the timer are initialized. It also shows how the aperiodic *NewTemp* signal is received and handled. It is important to make the aperiodic transitions inside the periodic process as short as possible, and, as in this example, try to make the event-related operations outside the process. The reason for this is that a transition cannot be preempted by another transition in the same process, and therefore the periodic transition must wait for the aperiodic to finish before it can execute.
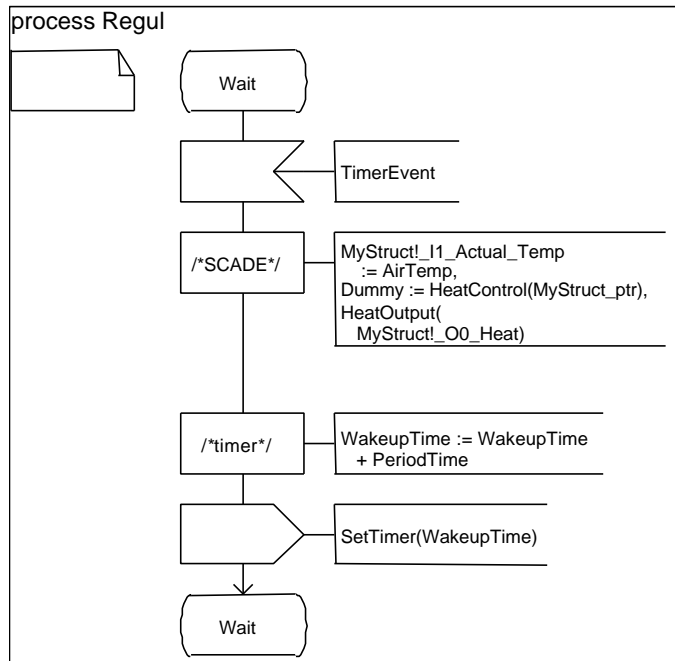
Figure 5.8: Periodic transition in Regul

The periodic transition in process *Regul* is triggered by the signal *TimerEvent* from the external timer. The *TimerEvent* signal is prioritized since that transition is more time-critical than the *NewTemp* transition.

The SCADE block is executed in three steps. The first step is to collect the air temperature with the external function *AirTemp*. The second step is to calculate the output to the heater with the SCADE defined function *HeatControl*. The third step is to set the new value for the heater with the external function *HeatOutput*.

After the SCADE-related part of the transition is executed, the timer must be set to a new time. In this example the timer value is increased with the period time and then sent to the external timer with the signal *SetTimer*.

The response time for the *TimeEvent* signal can be calculated from either one of the following three scenarios:

- The SDL system is idle:
  Response time = <time for the operating system to make a context switch> + <time for the SDL system to start the *TimerEvent* transition> + <time for *AirTemp*, *HeatControl*, and *HeatOutput*>

- The SDL system is executing the *Controller* process:
  Response time = <Response time for SDL system when it is idle> + <time for the SDL system to preempt *Controller* and make a context switch>

- The SDL system is executing the *Regul* process (*NewTemp* transition): Response time = <Response time for SDL system when it is idle> + <Time to finish the *NewTemp* transition>

There will be no signals sent to the system before the system has finished the initialization in this example, and therefore no response time for that scenario is calculated. If it is possible to send signals to the system during startup, the initialisation time must be added to the response time.

5.5 Scheduling

Scheduling the processes in the system is interesting when the system contains more than one process. SDL only supports FIFO scheduling. FIFO scheduling means that the kernel always chooses the process that has waited the longest to execute.

It is possible to use Rate-Monotonic scheduling in the SDL Suite. This scheduling is implemented with process priority. Rate-Monotonic scheduling is based on period time, where the process with the shortest period time has the highest priority. This scheduling requires that the user knows the period times and sets the priorities accordingly.

6 Improvement Suggestions

This chapter will present suggestions on how to improve the hard real-time aspects of the SDL tool. Ideas of how integration between SCADE and SDL can be supported will also be provided.

6.1 Connecting SCADE and SDL

This section will discuss how integration between SDL and SCADE could be supported. Three different approaches to make the integration will be presented.

6.1.1 Import SCADE into SDL

This is the first of the three suggested approaches to integrate SCADE and SDL. The approach is based on the working solution described in Section 5.1. This means that the SCADE function is imported and used in a SDL process. The import of the SCADE functions in SDL must be made much easier. Instead of compiling the SCADE node into a C function and then import the function with *CPP2SDL* and a *PR*-file, a reference to the SCADE node should be enough. By double-clicking on a SCADE-reference symbol in the SDL Process View, the SCADE node should pop up.

The SDL tool should show function and variable names in the structure containing the SCADE data. The names from SCADE could be presented in a menu in the Process View. The tool should also warn the user if the SCADE node is changed, and the SDL part of the program has to be updated.

The integration between SDL and SCADE in this approach has the advantage that the SCADE tool does not have to be changed at all from the users point of view, the integration is entirely made in the SDL suite. The changes should be fairly local in the SDL Suite since all integration is made in the Process View.

6.1.2 Map SCADE and SDL Signals in the SDL Tool

This approach is based on that it should be possible to describe a process behavior in ether SDL symbols or SCADE symbols. This means that when the developer creates a new process, he, or she, can choose to create either a SDL process or a SCADE process. This integration depends on the possibility to map the SCADE input and output, and SDL signals.

The example in Section 5.4 shows that there can be two kinds of signals to the SCADE defined process. Some of the signals only update the SCADE inputs while other signals results in both input update and execution of the SCADE node. The example also showed that sometimes it might be desirable to be able to connect a SCADE input directly to an external function.

The following example will show one possible way of making the integration of SCADE and SDL easier. This is an example with a heat controller where the user

can set the desired temperature as well as the two controller parameters *P* and *I*. *Actual_Temp* is read from a thermometer with the external function *AirTemp*. *Heat* will be calculated periodically defined by a period time or when *Wanted_Temp* is signaled. The *Heat* output will send a signal to the environment and call the external function *HeatOutput*.
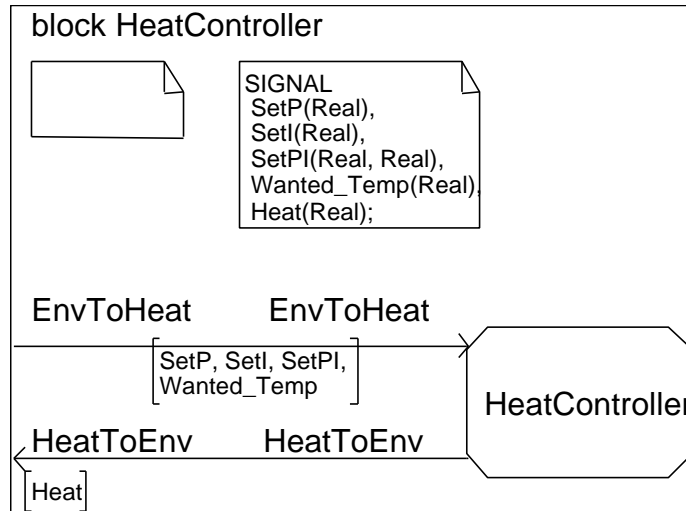
Figure 6.1: Block view of Heat Controller Example

Figure 6.1 shows the Block View of the *HeatController*, and the definition of the signals. When the Block View has been defined and the user opens the process *HeatController* a regular process view will come up.
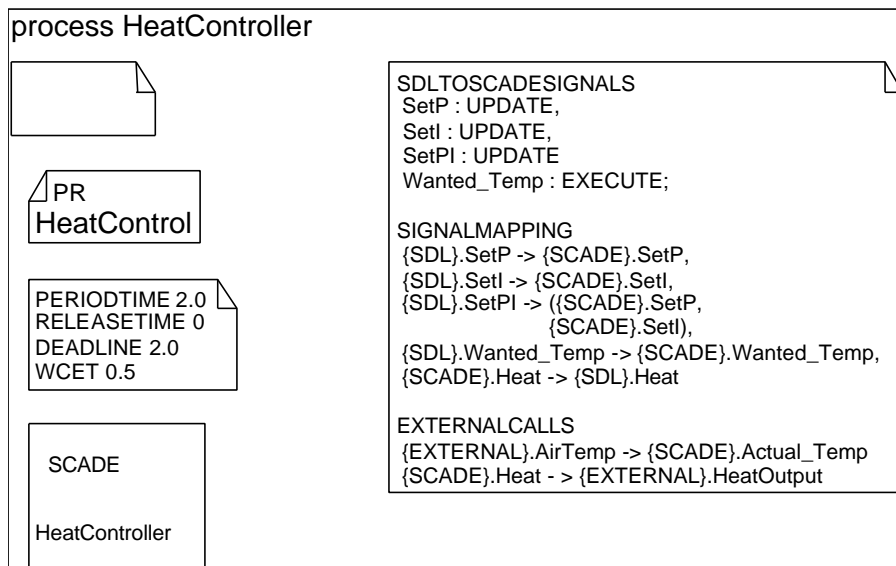
Figure 6.2: SDL to SCADE mapping in the process view.

The Process View in Figure 6.2 shows a PR reference to *HeatControl*, which is a file defining the external functions *AirTemp* and *HeatOutput*. The declaration block with *PERIODTIME* etc., defines the process behavior from a scheduling perspective. This block will be discussed in the Section 6.3.

24

The SCADE block is a new symbol indicating that the process behavior will be defined in SCADE. The user can open SCADE by double-clicking on this symbol. The large declaration block with *SDLTOSCADESIGNALS* defines how the SCADE defined parts shall be connected to SDL and the external functions. There are three different blocks in the definition:

- *SDLTOSCADESIGNALS*: Defines how the process reacts to a SDL signal. If the signal is defined as *UPDATE*, the signal will result in an update of one or more process variables. *EXECUTE* will first update process variables and then call the SCADE node for calculation. Note that an *EXECUTE* signal does not necessarily have to update any variables.

- *SIGNALMAPPING*: Defines which SDL signals that update the corresponding SCADE input variables, and which SCADE outputs that updates the corresponding SDL signals. Note that a signal with many values can update more than one SCADE input. This is done with signal *SetPI*, which will update both *P* and *I* input in SCADE. When *Heat* output is calculated in SCADE, the result will be sent with the SDL signal *Heat*.

- *EXTERNALCALLS*: Defines how SCADE input and output should be connected to external functions. In this example *AirTemp* should be read to the SCADE input *Actual_Temp* before the SCADE node is run. When SCADE output *Heat* is calculated the external function *HeatOutput* should be called with the result as parameter.

After defining the interface between SCADE and SDL, SCADE is opened and the SCADE node is defined which can be seen in Figure 6.3.
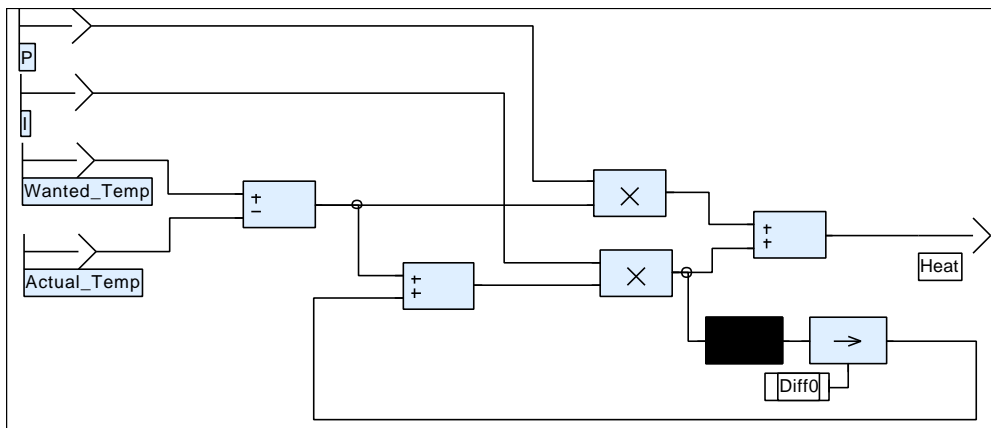
Figure 6.3: SCADE Process View

The main advantage of this approach is that the user only has to define the process behavior in SDL symbols or in SCADE symbols. The user does not have to work with pointers either, which the help in SDL warns for. The disadvantage is that the mapping code can become quite complex, and a user has to learn how to use this code. The lack of flexibility and the unfamiliar way of using SDL makes this approach less interesting than the previous approach.

6.1.3 Map SDL and SCADE Signals in the SCADE tool

An alternative solution is to define the input in SCADE where the input type can be selected in a property page. The input can be Updating, Executing or External Source. The Updating and the Executing Input behaves the same way as the UPDATE and EXECUTE inputs in the previous example. The External Source will call an external function for input. This function will be included in the property page for the input.
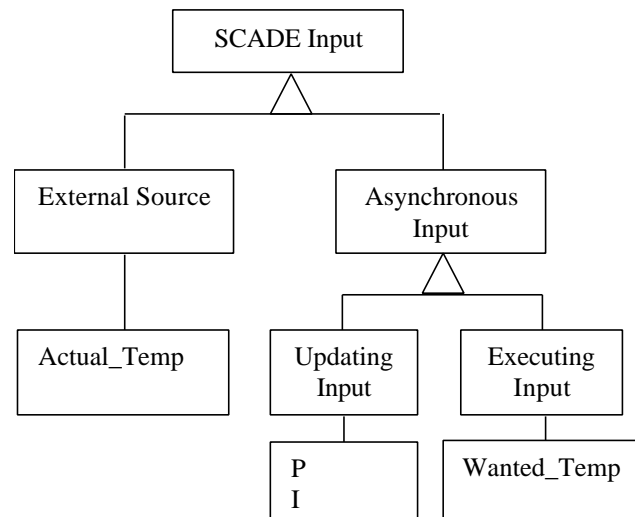
Figure 6.4: Different kinds of SCADE Input

Figure 6.4 shows the different inputs, and how the inputs in Figure 6.3 fit into the input tree.

This solution means that the Process View can be skipped and the SCADE tool can be started directly when accessing the process. This solution will be very hard to implement in the tools since both the SCADE Suite and the SDL Suite have to be changed. It is recommendable to minimize the effect on the existing tools, for a faster, more reliable and cheaper development of the tools.

6.2 Timers

Timers have been discussed in previous chapters, and the arguments for an external timer have been presented. The use of an external timer should be included as a macro in the Configuration Header File, *ml_mcf.h.*

#define XMK_USE_EXTERNAL_TIMER

When the macro is defined in the Configuration Header File all the timers in the system shall set an external timer. This can be done in two ways, one external timer for all internal timers, or one external timer for each internal timer. If only one external timer is used for the system, the scheduler must keep track of all the internal timers and put all the timer events in a list, from which the earliest timer event sets the external timer.

26

6.3 Scheduling

When a system contains more than one process, scheduling must be done. This section will provide scheduling algorithms that the SDL system should support. It will also discuss what is needed to be able to use these algorithms. Most information in the Scheduling chapter has been taken from [TSC00]

6.3.1 Periodic Process Scheduling

This section will discuss scheduling policies for periodic processes. There are many more policies than mentioned in this section, but these have been chosen because they are the most common scheduling policies.

- **Fixed Priority Preemptive Scheduling:** A process with higher priority preempts a process with lower priority. This scheduling policy can already be used in the SDL Suite.

- **Rate-Monotonic Scheduling:** The shorter the period time, the higher the priority. This is an optimal fixed priority policy, based on that the deadline and period time for a periodic task is the same. This policy can be implemented in the present SDL Suite using priority, but it would be a nicer solution if the user only had to define period time.

- **Deadline-Monotonic Scheduling:** Assumes that the deadline is a fixed point in time relative to the period start. The shorter the deadline, the higher the priority. This can be implemented in the present SDL Suite, but it would be nicer to define a deadline instead of priority.

- **Earliest Deadline First Scheduling:** This is a dynamic priority preemptive policy. The scheduler runs the process with the earliest deadline. A process with an earlier deadline preempts a process with a later deadline. This policy assumes that the deadlines for all the processes are known. The policy minimizes the maximum lateness of the tasks. In order to use this scheduling algorithm, deadline must be defined.

- **Least Slack Scheduling:** This scheduling policy uses dynamic-priority. The slack is calculated as the deadline minus the remaining worst-case execution time for the process to complete its execution. The process with the shortest slack will run first.

6.3.2 Aperiodic Process Scheduling

Since a real-time system often contains both periodic and aperiodic processes, the scheduling policy must be able to schedule both types of processes. One way of solving the scheduling problem is the Aperiodic Server Approach. The idea with this approach is to create a periodic task called Aperiodic Server and then let the aperiodic processes execute during the Aperiodic Servers scheduled time. The Aperiodic Server can be implemented in different ways.

- **Deferrable Server:** The Deferrable Server will have a period time T_{server} and an execution time C_{server}. This scheduling policy will reserve execution time every period, regardless if the reserved time is used or not.

- **Sporadic Server:** The Sporadic Server has a period time and an execution time like the deferrable server. The reserved execution time will not get replenished until T_{server} time after usage of reserved time. While the Deferrable server is easy to implement the Sporadic Server has higher schedulable utilization and is easier to analyze.
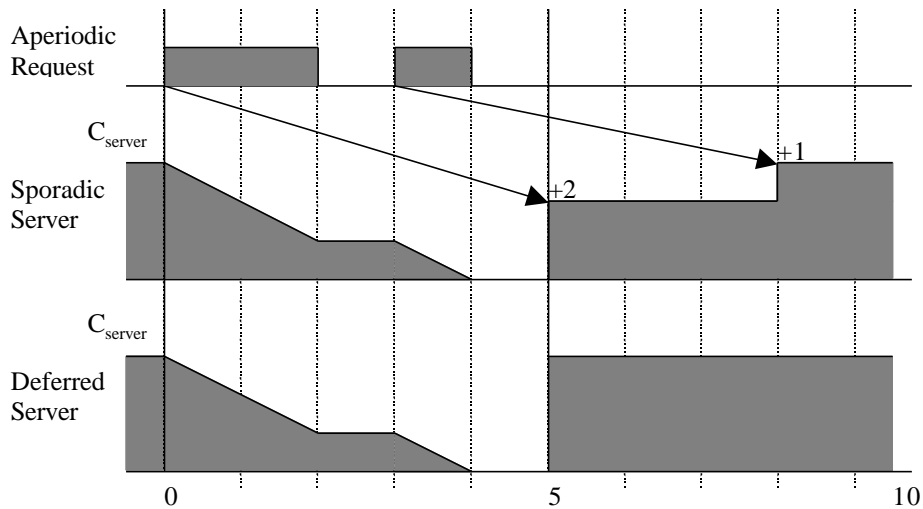
Figure 6.5: Sporadic and Deferrable Servers with Tserver = 5 and Cserver = 3

In Figure 6.5, an example with a Sporadic and a Deferrable Server is presented. The Figure shows how the Deferrable Server renews its execution time C_{server} in the beginning of a new period, while the Sporadic Server renews C_{server} one period time after the Server starts using execution time.

6.3.3 Process Attributes for Scheduling

The previous two sections showed that a number of process attributes are needed for the different scheduling policies. The interesting attributes are Worst Case Execution Time, or WCET, Period Time and Deadline. In some applications the Release Time is interesting for the scheduler to be able to make correct calculations of remaining execution time. [LHe99] suggests that the process attributes are declared as global variables for the process. Figure 6.6 shows how this can be done.
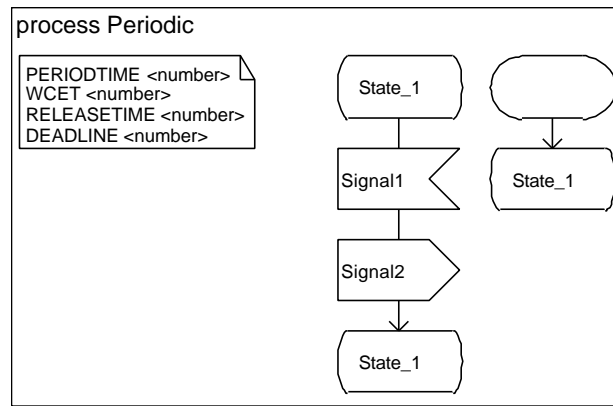
Figure 6.6: Periodic Process Attributes Definition

Both Period Time and Deadline are values that the user decides. WCET is the longest time a transition in the process can execute. It is very hard to calculate WCET, since a process can have many transitions and every transition can have more then one execution path. The easiest way to obtain WCET is to test the system and time all processes. Release Time should be the same for all processes, and should be very easy to obtain. The Release Time should be timed during tests.

7 Implementation of a Scheduling Algorithm

The last part of this Masters Thesis has been about developing a scheduling algorithm for the Cmicro kernel. This chapter will discuss how the algorithm was implemented, and also what should be done to make the scheduling more efficient and useful.

The chosen algorithm was *Earliest Deadline First* algorithm. The reason for this is that the *Earliest Deadline First* is a dynamic-priority algorithm and cannot be implemented with today's SDL tool, while Deadline-Monotonic and Rate-Monotonic scheduling can be done if the user defines priority correctly, see Section 6.3.1. Although this chapter only will discuss the *Earliest Deadline First* algorithm, other algorithms can be implemented in the same way.

To simplify the implementation of the scheduling algorithm, assumptions on that all processes have a defined a period time and a deadline have been made. This assumption is of course not applicable in many cases, but the scheduler is an example of how a scheduling algorithm can be introduced in the existing kernel.

The scheduling algorithm was built on the assumption that all processes have declared the variables *PERIODTIME*, *DEADLINE*, *NEXTPERIOD*, and *NEXTDEADLINE*, in the beginning of the variable declaration, and that the variables are declared in exactly that order, see Figure 7.1 in Section 7.1.

7.1 The Scheduled SDL System

A simple SDL system has been created, which was made to fit the scheduling algorithm. The system has two processes, each with a timer used to make a transition with a defined period time and deadline. The only differences between the two processes are the actual period time and deadline. Figure 7.1 shows how one of the processes is defined. The process will increase *NEXTPERIOD* with *PERIODTIME* every transition, and set the timer with *NEXTPERIOD*. *DEADLINE* and *NEXTDEADLINE* are used in the kernel to make the scheduling decision.

The compiler directive */*#Name* is used to make it possible to access the variables from the kernel. Without the */*#Name*-directive, the compiler will give the variables prefixes which are impossible to predict in the kernel.
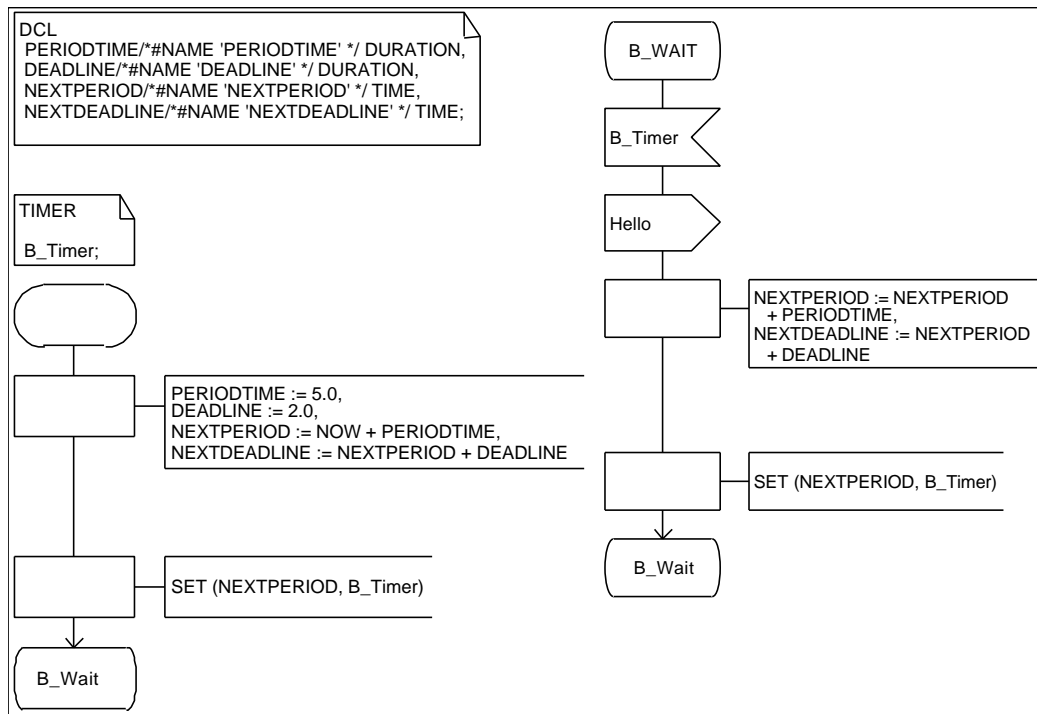
Figure 7.1: Periodic Process in SDL

7.2 Original Scheduling in the Cmicro Kernel

This section is an overview of how the Cmicro kernel works. The kernel runs in a loop, in which zero or one transition is executed each time. The kernel first checks if there are any signals from the environment, and puts the signals in the signal queue. The signal queue is global for all processes that the kernel handles. The second step in the loop is to check if any timer in the timer list has expired. The kernel puts the expired timers in the signal queue. The last step in the loop is to choose a signal to process. This is done in the function *ProcessSignal*.

ProcessSignal runs in a loop until one transition is made, or the signal queue is empty. This is done in three steps. The first step is to get the first signal in the signal queue. The second step is to check if the called process handles the signal, if not remove the signal and start the loop again. If the process handles the signal, the third step is to make the transition. The execution leaves the *ProcessSignal* loop when the transition is made.
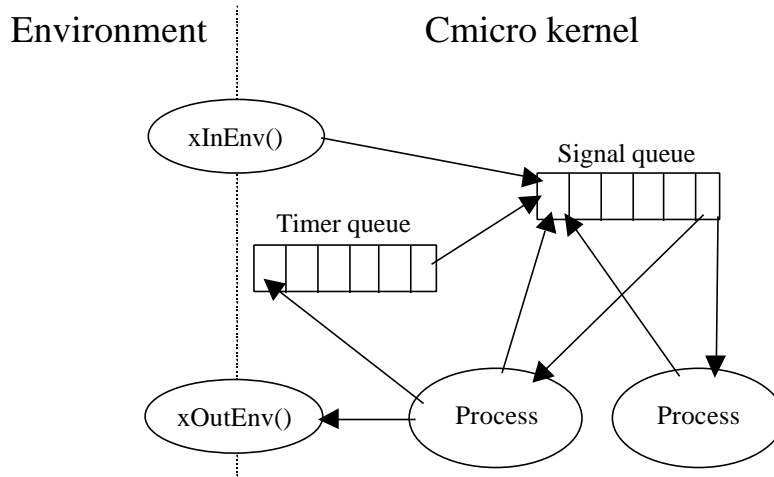
Figure 7.2: Timer queue and Signal queue in Cmicro

Figure 7.2 shows how signals are passed inside the Cmicro kernel and also between the environment and the kernel. The kernel calls *xInEnv* to recieve signals from the environment, and calls *xOutEnv* to send signals to the environment. Note that the kernel has one global signal queue and one global timer queue, which are used for all processes.

7.3 Changes in the Cmicro Kernel

The implementation of the new algorithm was made to make as few changes in the existing kernel as possible. This has resulted in changes in the function *ProcessSignal* in the *sche.c*-file, the new files *SchedulingAlgorithm.c* and *SchedulingAlgorithm.h*, see Appendix B and Appendix A, and a new function, *xmk_SetCurrentSignal*, in the queue-handling file *queu.c,* see Appendix C.

The decision of which signal that is going to be scheduled, in *ProcessSignal*, has been replaced with a call to the new scheduling function *xmk_GetNextScheduledProcess,* see Appendix D.

xmk_GetNextScheduledProcess can be found in the SchedulingAlgorithm-files. This function calls the defined scheduling algorithm function. The original SDL scheduling algorithm, *ClassicSDL*, will be called if no compiler flag is defined. The function *EarliestDeadlineFirst* will be called if the *XMK_EARLIEST_DEADLINE_FIRST* compiler flag is defined in the *Configuration Header File*.
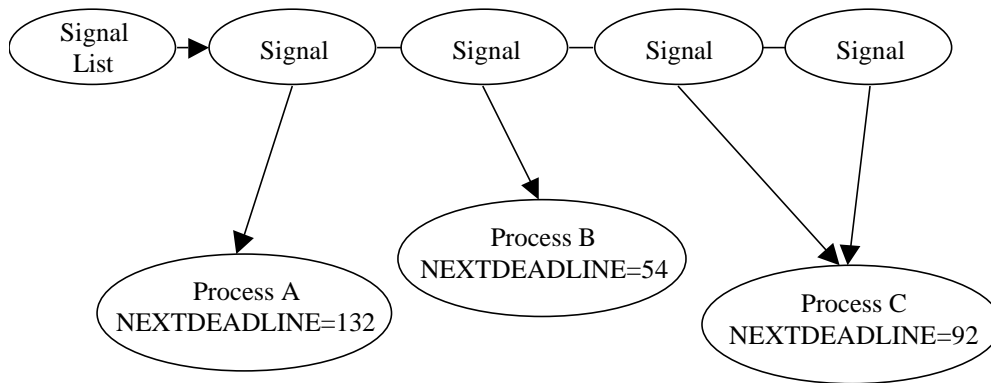
Figure 7.3: Signal queue with signals to processes A, B and C.

ClassicSDL will get the first signal in the signal queue. The selection part of the original *ProcessSignal* function has been moved to this function without changes. Figure 7.3 shows a signal queue with four signals to the processes *A*, *B* and *C*. The execution order will be *A, B, C,* and *C* with *ClassicSDL*.

EarliestDeadlineFirst goes through the signal queue and checks which of the signaled processes that has the earliest deadline. The structure *InstanceHeader* was defined to make it possible to access the instance variable *NEXTDEADLINE*. It is because of this mapping the instance variables must be in the correct order in the beginning of the variable declaration. With the queue in Figure 7.3, the execution order will be *B, C, C*, and *A* if *EarliestDeadlineFirst* is used.

Since the queue has a pointer to the selected message, the queue pointer has to be updated in *EarliestDeadlineFirst*-function. The reason for this is that the algorithm goes through the list and ends when the pointer has reached the end of the queue. *xmk_SetCurrentSignal* is called to set the pointer to the selected signal in the end of the *EarliestDeadlineFirst*-function.

7.4 Suggested Implementation Improvements

This section will present suggestions of how to improve the implementation of the scheduling algorithm.

The variables used by the kernel to schedule the periodic processes, in the implementation: *PERIODTIME*, *DEADLINE*, *NEXTPERIOD*, and *NEXTDEADLINE*, should be reserved words in the SDL Tool. These variables have to be reachable both from the kernel and the process. In this solution a boolean variable making it possible to check if the scheduling variables are defined is needed.

The schedulers preemption will not work properly together with the *Earliest Deadline First*-algorithm. It is therefore recommendable that the preemption part of the kernel is changed to work with the new algorithm as well. The problem with the preemption is that the kernel preempts based on priority. *Earliest Deadline First* is supposed to preempt if a process with earlier deadline wants to execute, priority is not used in this scheduling algorithm.

In order to schedule both periodic and aperiodic processes, it would be a good idea to have a special queue for the signals resulting in periodic transitions. The scheduler will then schedule the periodic transitions first and then the aperiodic. It is impossible to make an efficient decision whether a signal will result in a periodic transition or not, since there are no periodic timers and it is not defined whether a transition is periodic or not. One way of dealing with this problem is to look at all timers as periodic. This means that the kernel will put the resulting signal in the periodic queue when a timer expires. Periodic timers would be the best solution, making it possible to decide if a signal is periodic or not.

In order to make the selection of the executing process more efficient, it might be a good idea to sort the signal queue after deadlines when a new signal arrive. The advantage of sorting the signal queue is that the kernel does not have to go through the queue every time a scheduling decision is made.

8 Conclusion

This report has shown that the SDL and the SCADE tools work well together. Using SDL as a host language for SCADE is a good way of creating control systems with many processes. The connection between SCADE and SDL works, but could be better supported. A direct link between the tools is suggested.

Suggestions on how to improve the SDL tool from a real-time aspect have been presented. The SDL tool needs to improve the time management. Timers in the existing tools are not accurate enough. This paper suggests that external timers should be optional, creating better accuracy in time. Another suggestion to improve the SDL tool is to have scheduling as an option. Different periodic and aperiodic algorithms would strengthen the tool in a hard real-time aspect.

This report has shown that it is possible to introduce a new scheduling algorithm to the SDL Cmicro kernel. It is important though to remember that the algorithm has only been tested with a few simple test systems. There are probably a lot of problems with the new kernel, but this is only an example of how an algorithm can be introduced.

9 References

[CaSe]   J.-L.Camus, T.L.Sergent. Combining Telecom Techniques with
            Control Engineering Techniques for Distributed Control Systems

[EHS97]   J.Ellsberger, D.Hogrefe, A.Sarma. SDL Formal Object-oriented
            Language for Communicating Systems. Great Britain, 1997.
            ISBN: 0-13-632886-5.

[Hal93]   N.Halbwachs. Synchronous Programming of Reactive Systems.
            Netherlands 1993. ISBN: 0-7923-9311-2

[LHe99]   L.Helmestam, Tools for real-time systems. Master thesis,
            Department of Computer Engineering Mälardalen University,
            1999.

[SCADE]   Help in Telelogic Tau SCADE, Telelogic.

[SDL41]   Help in Telelogic Tau SDL Suite 4.1, Telelogic.

[TSC00]   The Concise Handbook Of Real-Time Systems, Version 1.1.
            TimeSys Corporation, 1999.

10 Glossary

Bare Integration – Cmicro Integration with environment without operating system

Block – Part of SDL language, see Section 3.1

Block View – Window in the SDL tool for editing blocks

Earliest Deadline First – periodic scheduling policy, see Section 6.3.1

Deadline – Point in time when a task must have finished its execution

Deadline-Monotonic – periodic scheduling policy, see Section 6.3.1

Deferrable Server – aperiodic scheduling policy, see Section 6.3.2

FIFO – First In First Out is a scheduling policy

ITU – International Telecommunication Union

Least Slack Scheduling – periodic scheduling policy, see Section 6.3.1

Light Integration (Cadvanced) – Integration with environment using SDL kernel

Light Integration (Cmicro) – Integration with environment using SDL kernel

Period Time – Time interval between two executions of a periodic task

Preemption – The scheduler end a process execution before it is finished

Process - Part of SDL language, see Section 3.1

Process View - Window in the SDL tool for editing processes

Rate-Monotonic – periodic scheduling policy, see Section 6.3.1

Response Time – The time it takes the system to create an output when an input is presented.

RTOS – Real-Time Operating System

SCADE – Safety Critical Applications Development Environment

SDL – Specification Description Language

Sporadic Server – aperiodic scheduling policy, see Section 6.3.2

System – Part of SDL language, see Section 3.1

System View – Window in the SDL tool for editing systems

Tight Integration – Cadvanced Integration with environment using Operating System primitives

Appendix A: SchedulingAlgorithm.h

```
/*+MHDR*/
/*-MHDR*/
/*
 +------------------------------------------------------------------------+
 |                                                                        |
 | Copyright by Telelogic AB 1993 - 2000                                  |
 |                                                                        |
 | This Program is owned by Telelogic and is protected by national        |
 | copyright laws and international copyright treaties. Telelogic          |
 | grants you the right to use this Program on one computer or in          |
 | one local computer network at any one time.                            |
 | Under this License you may only modify the source code for the purpose  |
 | of adapting it to your environment. You must reproduce and include      |
 | any copyright and trademark notices on all copies of the source code.   |
 | You may not use, copy, merge, modify or transfer the Program except as  |
 | provided in this License.                                              |
 | Telelogic does not warrant that the Program will meet your              |
 | requirements or that the operation of the Program will be              |
 | uninterrupted and error free. You are solely responsible that the      |
 | selection of the Program and the modification of the source code        |
 | will achieve your intended results and that the results are actually    |
 | obtained.                                                              |
 |                                                                        |
 +------------------------------------------------------------------------+
*/
/*
 +------------------------------------------------------------------------+
 | Functionname : GetNextScheduledProcess                                 |
 +------------------------------------------------------------------------+
 |                                                                        |
 | Description  : Return true if any process has been scheduled.          |
 |                This function search through the signal list and check which |
 |                receiving process shall be scheduled next. Returns pointer |
 |                to xPID and pointer to a pointer to message of type       |
 |                xmk_T_MESSAGE                                            |
 |                                                                        |
 | Parameter    : xPID * , xmk_T_MESSAGE **                               |
 |                                                                        |
 | Return       : unsigned char                                          |
 |                                                                        |
 +------------------------------------------------------------------------+
*/

#ifndef __SCHEDULINGALGORITHM_H_
#define __SCHEDULINGALGORITHM_H_

#include ml_typ.h

unsigned char xmk_GetNextScheduledProcess( xPID * p_ScheduledPID, \
        xmk_T_MESSAGE ** p_ScheduledMessage);

#ifdef XMK_EARLIEST_DEADLINE_FIRST
```

```
/*
+-----------------------------------------------------------------------+
|  Functionname : EarliestDeadlineFirst                                 |
+-----------------------------------------------------------------------+
|                                                                       |
|  Description  : Return true if any process has been scheduled.        |
|                 This function search through the signal list and check which |
|                 receiving process shall be scheduled next. Returns pointer   |
|                 to xPID and pointer to a pointer to message of type    |
|                 xmk_T_MESSAGE                                         |
|                                                                       |
|  Parameter    : xPID * , xmk_T_MESSAGE **                             |
|                                                                       |
|  Return       : unsigned char                                         |
|                                                                       |
+-----------------------------------------------------------------------+
*/

  unsigned char EarliestDeadlineFirst(xPID * p_ScheduledPID, \
     xmk_T_MESSAGE ** p_ScheduledMessage);

#else /* Old scheduling, FIFO or priority preemptive

/*
+-----------------------------------------------------------------------+
|  Functionname : ClassicSDL                                            |
+-----------------------------------------------------------------------+
|                                                                       |
|  Description  : Return true if any process has been scheduled.        |
|                 This function search through the signal list and check which |
|                 receiving process shall be scheduled next. Returns pointer   |
|                 to xPID and pointer to a pointer to message of type    |
|                 xmk_T_MESSAGE                                         |
|                                                                       |
|  Parameter    : xPID * , xmk_T_MESSAGE **                             |
|                                                                       |
|  Return       : unsigned char                                         |
|                                                                       |
+-----------------------------------------------------------------------+
*/

  unsigned char ClassicSDL(xPID * p_ScheduledPID, \
        xmk_T_MESSAGE ** p_ScheduledMessage);
#endif
#endif
```

Appendix B: SchedulingAlgorithm.c

```
/*+MHDR*/
/*-MHDR*/
/*
+------------------------------------------------------------------------------+
|                                                                              |
| Copyright by Telelogic AB 1993 - 2000                                        |
|                                                                              |
| This Program is owned by Telelogic and is protected by national              |
| copyright laws and international copyright treaties. Telelogic               |
| grants you the right to use this Program on one computer or in               |
| one local computer network at any one time.                                  |
| Under this License you may only modify the source code for the purpose       |
| of adapting it to your environment. You must reproduce and include           |
| any copyright and trademark notices on all copies of the source code.        |
| You may not use, copy, merge, modify or transfer the Program except as       |
| provided in this License.                                                    |
| Telelogic does not warrant that the Program will meet your                   |
| requirements or that the operation of the Program will be                    |
| uninterrupted and error free. You are solely responsible that the           |
| selection of the Program and the modification of the source code             |
| will achieve your intended results and that the results are actually         |
| obtained.                                                                    |
|                                                                              |
+------------------------------------------------------------------------------+
*/

#ifndef __SCHEDULINGALGORITHM_C_
#define __SCHEDULINGALGORITHM_C_

/*+IMPORT*/
/*==================== I M P O R T ================================*/
#include ml_typ.h
#include ml_err.h

/*------------------- Variables   ----------------------------------------*/
extern XCONST XPDTBL xmk_ROM_ptr xPDTBL[] ; /* <root-process-table>        */

/*================================================================*/
/*-IMPORT*/

/*------------------- Constants,  Macros ----------------------------------*/

/*
** Internal Value(running proc-type)
*/
#undef  RUN_PROC
#define RUN_PROC        EPIDTYPE(xRunPID)

/*
** Internal Value(running proc-inst)
*/
#undef  RUN_INST
#define RUN_INST        EPIDINST( xRunPID)
```

```
/*------------------- Typedefinitions    --------------------------------*/
/*****************************************************************
*Struct used to map the first variables in the instance struct
*defined in components.c
*****************************************************************/

#ifdef XMK_EARLIEST_DEADLINE_FIRST
  typedef struct {
    PROCESS_VARS
    SDL_Duration PERIODTIME;
    SDL_Duration DEADLINE;
    SDL_Time NEXTPERIOD;
    SDL_Time NEXTDEADLINE;
  } InstanceHeader;
#endif

/*------------------- Functions    --------------------------------------*/

/*------------------- Variables    -------------------------------------*/

/*
+------------------------------------------------------------------------+
|  Functionname : GetNextScheduledProcess                                |
+------------------------------------------------------------------------+
|                                                                        |
|  Description  : Return true if any process has been scheduled.         |
|               This function search through the signal list and check which |
|               receiving process shall be scheduled next. Returns pointer |
|               to xPID and pointer to a pointer to message of type       |
|               xmk_T_MESSAGE                                             |
|                                                                        |
|  Parameter    : xPID * , xmk_T_MESSAGE **                              |
|                                                                        |
|  Return       : unsigned char                                          |
|                                                                        |
+------------------------------------------------------------------------+
*/

unsigned char xmk_GetNextScheduledProcess( xPID * p_ScheduledPID, \
        xmk_T_MESSAGE ** p_ScheduledMessage)
{
  #ifdef XMK_EARLIEST_DEADLINE_FIRST
    return  EarliestDeadlineFirst (p_ScheduledPID, p_ScheduledMessage);
  #else
    return  ClassicSDL (p_ScheduledPID, p_ScheduledMessage);
  #endif
}
```

```
#ifdef XMK_EARLIEST_DEADLINE_FIRST

/*
+--------------------------------------------------------------------------------------+
|  Functionname : EarliestDeadlineFirst                                                |
+--------------------------------------------------------------------------------------+
|                                                                                      |
|                                                                                      |
|  Description  : Return true if any process has been scheduled.                       |
|                 This function search through the signal list and check which         |
|                 receiving process shall be scheduled next. Returns pointer           |
|                 to xPID and pointer to a pointer to message of type                  |
|                 xmk_T_MESSAGE                                                        |
|                                                                                      |
|  Parameter    : xPID * , xmk_T_MESSAGE **                                            |
|                                                                                      |
|  Return       : unsigned char                                                       |
|                                                                                      |
+--------------------------------------------------------------------------------------+
*/

EarliestDeadlineFirst (xPID * p_ScheduledPID, \
        xmk_T_MESSAGE ** p_ScheduledMessage)
{
  XPDTBL xmk_ROM_ptr p_ReceiverProcess  ;
  xINSTD xmk_RAM_ptr  pRunData;    /* Pointer to currently running     */
                                   /* processinstance - data           */
  xmk_T_MESSAGE xmk_RAM_ptr p_Message;              // Pointer to current messages in
                                                    // message list
  InstanceHeader * p_CurrentHeader;
  xPID xRunPID;                            // Current process PID
  SDL_Time CurrentDeadline;                // Current process Deadline
  SDL_Time EarliestDeadline;               // Earliest Deadline for the called processes

/********************************************************************* INIT SIGNAL POINTER */

  /*
  **   load pointer to signal with first signal in the queue
  */
  p_Message = xmk_FirstSignal ();

  EarliestDeadline = -1.0;

/********************************************* PROCESS FIRST PROCESSABLE SIGNAL */

  while( p_Message != (xmk_T_MESSAGE xmk_RAM_ptr) NULL )
  {
/********************************************************* GET SIGNALED PROCESS ID */
    #ifndef XMK_USE_RECEIVER_PID_IN_SIGNAL
    /*
    ** Set new active process-PID
    */

    XMK_BEGIN_CRITICAL_PATH;
    xRunPID = xRouteSignal (p_Message->signal);
    XMK_END_CRITICAL_PATH;

/***************************************************************** CHECK IF PID IS VALID */
```

```c
   #ifdef XMK_USE_MAX_ERR_CHECK
    if (xRunPID == xNULLPID)
    {

      #if defined (XFREESIGNALFUNCS) && defined \
         (XMK_USED_SIGNAL_WITH_DYN_PARAMS)
        xmk_ReleaseSignalParameter(p_Message);
      #endif
      xmk_RemoveCurrentSignal ();
      /*
       ** Check next signal in queue
       */
      p_Message = xmk_NextSignal ();
      continue;
    }
   #endif
  #else
/*********************************************************** GET SIGNALED PROCESS ID */

   /*
    **   Set new active process-PID
    */
   XMK_BEGIN_CRITICAL_PATH;
   xRunPID = p_Message->rec;
   XMK_END_CRITICAL_PATH;
  #endif /* ... ifndef XMK_USE_RECEIVER_PID_IN_SIGNAL */

/**************** CHECK IF DEADLINE IS SHORTER THAN PREVIOUS PROCESSES */

  /*
  ** Get current process type
  */

  p_ReceiverProcess = xPDTBL [ RUN_PROC ];

  /*
  ** Get current instance structure
  */

  #ifndef XMK_USED_ONLY_X_1
   pRunData  = (xINSTD xmk_RAM_ptr)(p_ReceiverProcess->pInstanceData \
                     + ( p_ReceiverProcess->DataLength * ( RUN_INST ) ) );
  #else
   pRunData  = (xINSTD xmk_RAM_ptr)(p_ReceiverProcess->pInstanceData);
  #endif

  p_CurrentHeader = (InstanceHeader *) pRunData;
  CurrentDeadline = (SDL_Time) p_CurrentHeader->NEXTDEADLINE;

  /*
  ** Check if current process has the earliest deadline
  */

  if (EarliestDeadline >= 0)
  {
   if (CurrentDeadline < EarliestDeadline)
   {
     EarliestDeadline = CurrentDeadline;
     *p_ScheduledPID = xRunPID;
     *p_ScheduledMessage = p_Message;
```

```
      }
    }
    else
    {
      EarliestDeadline = CurrentDeadline;
      *p_ScheduledPID = xRunPID;
      *p_ScheduledMessage = p_Message;
    }

    p_Message = xmk_NextSignal();
  } /* END WHILE */

/****************************************Update queue pointer and reset temporary poiners */

  /* Set pointer in queue to the scheduled signal */
  xmk_SetCurrentSignal(*p_ScheduledMessage);

  /* Set pointers to null */
  p_Message = (xmk_T_MESSAGE xmk_RAM_ptr) NULL;
  p_ReceiverProcess = NULL;
  pRunData = NULL;
  p_CurrentHeader = NULL;

  if (EarliestDeadline >= 0)
  {
    return 1;
  }
  else
  {
    return 0;
  }
} /* END OF FUNCTION */

#else /* Old scheduling, FIFO or priority preemptive

/*
+---------------------------------------------------------------------------------+
| Functionname : ClassicSDL                                                       |
+---------------------------------------------------------------------------------+
|                                                                                 |
| Description  : Return true if any process has been scheduled.                   |
|                This function search through the signal list and check which     |
|                 receiving process shall be scheduled next. Returns pointer      |
|                to xPID and pointer to a pointer to message of type              |
|                xmk_T_MESSAGE                                                     |
|                                                                                 |
| Parameter    : xPID * , xmk_T_MESSAGE **                                        |
|                                                                                 |
| Return       : unsigned char                                                    |
|                                                                                 |
+---------------------------------------------------------------------------------+
*/

ClassicSDL (xPID * p_ScheduledPID, xmk_T_MESSAGE ** p_ScheduledMessage)
{
  unsigned char Result;
  xPID xRunPID;

  Result = 1;
  /*
```

44

```
  ** Use a Routing function, if the signal in the queue contains
  ** no receiver-PID. The user has to implement the function
  ** anywhere; a template is given in mk_user - module.
  */
 #ifndef XMK_USE_RECEIVER_PID_IN_SIGNAL
  /*
  **  Set new active process-PID
  */

/************************************************************* GET SIGNALED PROCESS ID */
   XMK_BEGIN_CRITICAL_PATH;
   xRunPID = xRouteSignal ((*p_ScheduledMessage)->signal);
   XMK_END_CRITICAL_PATH;

/*************************************************************** CHECK IF PID IS VALID */

   #ifdef XMK_USE_MAX_ERR_CHECK
    if (xRunPID == xNULLPID)
    {
      //ErrorHandler (ERR_N_xRouteSignal);

      #if defined  (XFREESIGNALFUNCS) && defined \
         (XMK_USED_SIGNAL_WITH_DYN_PARAMS)
       xmk_ReleaseSignalParameter(*p_ScheduledMessage);
      #endif
      xmk_RemoveCurrentSignal ();
      /*
      **  Leave signal-handling ( leave while-loop )
      */
      *p_ScheduledMessage = NULL;
      Result = 0;
    }
   #endif
  #else

/************************************************************* GET SIGNALED PROCESS ID */

  /*
  **  Set new active process-PID
  */
   XMK_BEGIN_CRITICAL_PATH;
   xRunPID = (*p_ScheduledMessage)->rec;
   XMK_END_CRITICAL_PATH;
 #endif /* ... ifndef XMK_USE_RECEIVER_PID_IN_SIGNAL */

 *p_ScheduledPID = xRunPID;
 return Result;
}
#endif
#endif
```

Appendix C: xmk_SetCurrentSignal

```
/*+FHDR E*/
/*
+---------------------------------------------------------------------+
| Functionname : xmk_SetCurrentSignal                                 |
+---------------------------------------------------------------------+
|                                                                     |
| Description  :                                                      |
| Set the queue pointer to point at p_Message                         |
|                                                                     |
| Parameter    : -   xmk_T_MESSAGE *                                  |
|                                                                     |
| Return       : -                                                    |
|                                                                     |
+---------------------------------------------------------------------+
*/
/*-FHDR E*/

#ifndef XNOPROTO
 void xmk_SetCurrentSignal( xmk_T_MESSAGE * p_Message )
#else
 void xmk_SetCurrentSignal ( p_Message )
 xmk_T_MESSAGE xmk_RAM_ptr p_Message ;
#endif

{
 XMK_BEGIN_CRITICAL_PATH;
 XMK_CURRENTSIGNAL = XMK_QUEUE;
 XMK_END_CRITICAL_PATH;

 while ( (XMK_CURRENTSIGNAL != (T_E_SIGNAL xmk_RAM_ptr) NULL ) \
         && (( xmk_T_MESSAGE xmk_RAM_ptr) XMK_CURRENTSIGNAL != \
         p_Message))
 {
  XMK_BEGIN_CRITICAL_PATH;
  XMK_CURRENTSIGNAL = (XMK_CURRENTSIGNAL)->next;
  XMK_END_CRITICAL_PATH;
 }
}
```

Appendix D: xmk_ProcessSignal

The following code is the part of ProcessSignal where GetNextScheduledProcess is called:

```
...
p_tempMessage = &p_Message;
p_xRunPID = &xRunPID;
while( p_Message != (xmk_T_MESSAGE xmk_RAM_ptr) NULL )
{
 if (!xmk_GetNextScheduledProcess (p_xRunPID, p_tempMessage))
 {
  p_Message = NULL;
  continue;
  }
...
```