Development of a Runtime Environment for SDL Systems on Resource-Limited Platforms

M. Krämer, T. Kuhn

Technical Report 345/05

Development of a Runtime Environment for SDL Systems on Resource-Limited Platforms

M. Krämer, T. Kuhn

Computer Science Department, University of Kaiserslautern, Kaiserslautern, Germany
{kraemer,kuhn}@informatik.uni-kl.de

Technical Report 345/05

Computer Science Department
University of Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany

Technical Report

Development of a Runtime Environment for SDL Systems on Resource-Limited platforms

M. Krämer, T. Kuhn

[kraemer, kuhn]@informatik.uni-kl.de

Abstract

This report describes our ongoing work of creating a runtime platform for SDL systems on resource limited platforms. It mainly focuses the connection of generated SDL systems with a generic environment through the SDL Environment Framework for micro controllers, called µSEnF. The µSEnF enables SDL systems to communicate through a mostly hardware independent interface with the native hardware. This approach keeps the SDL systems portable, since no hardware dependent functions are contained within the SDL systems. It also keeps µSEnF portable, since its interaction points with generated SDL systems and operating systems are well defined. This way, only relevant parts of µSEnF must be changed when it is ported to another runtime platform, or when it should be used with a different code generator. Additionally, this report presents our first experiences on creating code for the MicaZ motes from crossbow, a platform that is used for very small applications within the domain of ambient intelligence and sensor networks.

1. Introduction

1.1. Motivation

Embedded systems become more important every day, so also the software that is running on these systems becomes more important. Especially in the still young domain of Ambient Intelligence Systems [AHS02], hardware platforms are expected to become smaller and more energy efficient, yielding in platforms with severe resource limitations. Since the costs of software systems currently are mainly controlled by money spent for maintenance, reengineering and debugging – also caused by the lack of available documentation – new methodologies have to be defined for developing software. One of these methodologies is the model driven development process. There are many methodologies how model driven development can be implemented in a specific domain, one possibility is the model driven development process with SDL that has been described in [FG05a].

This report presents our current work of creating and implementing a runtime platform for SDL systems on resource-limited platforms. Our runtime platform enables the usage of our model driven development process even on platforms with only very scarce resources. Model driven development on these platforms is a challenging task because applications are created with a theoretical model in mind, generally based on finite state machines. These models usually do not consider a limited amount of resources, for example for signal input queues. Mapping these theoretical models to resource limited hardware can cause the application to become unreliable because signals from one process to another one might be dropped. So there is a need not to only provide a runtime platform that is capable of executing models on these platforms, but also the reliable execution of these applications must be guaranteed. This includes hiding the limitations on queue size to the user or at least to provide methodologies to overcome this problem.

This report presents the current state of our work to implement an execution platform for SDL systems, and a first evaluation of the possibilities with this approach as well as the further work that has to be done in this area.

1.2. Domain description

In the embedded systems domain, especially when considering Ambient Intelligence systems, multiple specific requirements have to be considered. These are limited processing and memory resources, very limited energy resources – both constraints make the code generation and the generation of a runtime platform to execute this code for these Platforms a very challenging Task. Also real-time and reliability aspects have to be considered. These systems are usually expected to react reliably to a given situation within a defined period of time.

1.3. Our goals

For being able to perform model driven development on any platform, a runtime engine for executing these models has to be implemented. This is a challenging task, because the targeted hardware provides only very scarce resources. Also the domain specific requirements like reliability and the real-time abilities have to be considered. So our detailed goals for this ongoing work are the following:

- Execute SDL systems on resource limited ambient network nodes:

Provide a platform for executing SDL systems on resource limited ambient network nodes. Evaluate possible code generators, possible operating systems, whether automatic code generation is possible and the efficiency of the generated code. Also possible scheduling and memory management strategies should be evaluated and eventually be integrated into the SDL platform if necessary.

- Provide a generic hardware interface to common hardware on these nodes:
  Identify necessary hardware that are common to all ambient network nodes and provide a methodology for integrating new hardware into the platform in a documented manner.

- Evaluate the necessary partitioning:
  Ideally, all of the intelligence of the communication system should be implemented in SDL while the more generic tasks may be performed in the operating system, if the SDL platform is not accurate or fast enough to accomplish these tasks. Therefore, necessary interfaces are to be developed, based on experience from other projects, or based on literature studies.

- Evaluate the timing characteristics:
  The timing characteristics of the platform should be evaluated including scheduling delay and the jitter that must be calculated for receiving signals or timers. Achievable timing capabilities of such a platform need to be evaluated. We expect that some predictable timing characteristics can be achieved when performing a more tight integration of the SDL virtual machine with the used operating system as it is done in this work.

- Evaluate the achievable reliability:
  The platform to be developed must be reliable - also with respect to the limited memory resources and specific properties of the SDL language. Therefore, workarounds for the theoretical model of infinite SDL queues must be developed to keep the behavior of the system predictable – even under conditions with high signal load.

- Integrate into development- and simulation process:
  There should be a possibility to integrate the platform into our model-driven development- and simulation process.

1.4. Scope of this report

This is a report describing our work in progress. Currently, we have a platform capable of executing SDL systems on MicaZ motes, using an Atmel ATMega128L micro controller. We consider this platform as a first step in the direction of generating code from SDL systems, because the integration of the SDL system, mainly the scheduling of transitions and the realization of timers on the micro controller hardware is still very basic. This report presents the current state of our work, the basic structure of our runtime platform, its capabilities and an outlook on our future work plans with SDL systems on micro controllers.

1.5. Related work

There has been already some effort of integrating SDL code in embedded systems, either by integrating existing code generators with real-time operating systems, or by providing manual transformation schemes. This section will briefly survey the existing work in this area.

- Real-Time developer studio [Pra] is an implementation of SDL-RT [SDLRT], an extension of the SDL language for real time purposes. Some of the operating systems

supported by the Real-Time developer studio are capable of being executed on resource limited nodes like the MicaZ platform.

- [DRDK04] proposes the use of manual transformation patterns for transforming SDL systems into native code for TinyOS [HSWHCP00].
- Telelogics SDL Suite [Tel] offers code generation facilities for a variety of platforms supported by different code generators. While the CAdvanced code generator aims at generating code for larger embedded platforms like PC-Style hardware or XScale architectures, the CMicro code generator generates code for executing on resource limited platforms. Different types of integrations with the operating system are available and can be manually adapted.
- Contrast is a new SDL runtime environment that is developed at the University of Kaiserslautern. Although it looks very promising, it is currently far away of being usable on resource limited node platforms due to its memory requirements.
- Cinderella SDL [Cin] with the available code generators Cinderella SITE and Cinderella SLIPPER is capable of generating C++ code (if using SITE) or C code (if using SLIPPER). From the available information, it seems that the generated code from SLIPPER is not tightly integrated with operating systems, but provides a portable SDL virtual machine written entirely in C.
- The SDL operating system “Reflex”, developed at the Agder University College as a master thesis [WT04], is an entirely new SDL runtime environment, for which currently is no compiler available yet. The kernel offers the necessary features for mapping most SDL constructs to native functions of the operating system.
- In [DSH99], the possibility of generating VHDL specifications and C code out of SDL specifications is presented. It is shown, that time critical parts of SDL systems can eventually be implemented in hardware – directly generated from SDL specifications if software solutions are not efficient enough. Unfortunately, a performance analysis of the generated systems is missing in the paper.
- In [DZM01], the CAdvanced code generator of Telelogic TAU was used for generating code from a SDL specification. A tight integration with the Virtuoso RTOS was performed. Unfortunately, as with [DSH99], the timing characteristics of the resulting system were not published in the paper.
- In [ADLPT99] proposes the use of real-time temporal logic for integrating non-functional aspects like timing into a SDL system. A new, predictable, execution model is proposed which should enable the developers of performing analysis of their specifications. The timing behavior prediction is based on mapping the SDL specification to a transition graph, and performing a schedulability analysis afterwards.

Although some of the works mentioned above use real-time operating systems, no platform is capable of guaranteeing maximum delay and jitter values for response times and timers. Depending on the system load it might be possible that those guarantees are beyond reach for all processes. However, our goal is to offer predictable timing to at least a subset of SDL processes under specific conditions. A SDL system could consist then of a set of time critical processes and of a set of ordinary SDL processes.

Another important fact that also did rise during the first experiments with our runtime platform is its reliability. Especially when interfacing to communication hardware using a very low level interface, it is possible that message queues become filled. Then, some signals must be dropped – possibly resulting in unexpected behavior of the SDL system. This problem must also be investigated when developing a runtime platform for the ambient intelligence domain.

1.6. Structure of this work

The following sections are structured as following: Section 2 describes the requirements that have been identified for the runtime platform. Section 3 documents the design of the runtime platform, Section 4 highlights implementation details. In Section 5, the platform is evaluated. Section 6 summarizes conclusions and Section 7 points out further research areas.

2. Requirements and challenges

In this section, the specific requirements to a runtime platform for SDL systems in the context of ambient intelligence systems are evaluated. Ambient network nodes usually are equipped with low-power hardware. This includes a micro controller, some communication hardware, interfaces to sensors or general purpose I/O and eventually wireless communication hardware. Since our domain is the development of communication protocols for wireless Ad-Hoc networks, we did only consider sensor network hardware that is equipped with a transceiver chip for wireless communication.

2.1. Survey on available platforms

In the sensor network and in the ambient intelligence domain, mainly two platforms are available for research:

- The particles developed at the University of Karlsruhe:
  The particle computers, developed by the University of Karlsruhe are a quite new sensor network platform. The nodes are equipped with PIC micro controllers, interfaces for sensor hardware, serial data storage and a chip that provides a unique serial id. The used processor is a PIC18F6720 with 128 Kbytes of flash-ram for program code and 4 Kilobytes memory for data. Unfortunately, there are currently no nodes available for buying that contain the CC2420 transceiver chip – a chip that would provide a 802.15.4 compatible physical layer. Although such a particle is planned, it is not yet available. Also, there is not much documentation on the used operating system available.

- The Mica motes created by the University of Berkley:
  The Mica family is equipped with Atmel ATMega micro controllers, transceiver hardware, two RS232 ports and interfaces for sensor hardware. Furthermore, like on the particles, serial data storage is available and every node contains a chip with a unique ID. The Atmel ATMega128L has 128Kbytes flash-ram and 4Kbytes of data ram, so generating code for this platform from specifications will be a challenging task. The Mica nodes are operated using TinyOS; an operating system designed for very small devices and can be programmed by using the Gnu C Compiler or by using nc, an extension of the C language for resource limited platforms. It should be noted that also the Mica motes exist in multiple versions – we decided to use the newest MicaZ motes that are equipped with an 802.15.4 compatible transceiver chip – the CC2420. Since the CC2420 only provides the physical interface, it is possible to specify the Mac layer entirely in software, which is an excellent base for our research plans.

These two platforms are the platforms, that are most commonly used in research and industry. Because of availability issues and because of the available open operating system, we decided to favour the MicaZ nodes for the development of our SDL runtime environment. The CC2420 transceiver chip offers an 802.15.4 compatible physical layer which is a promising

technology in the emerging research area of sensor networks. Furthermore, the processors used on the Mica platforms have more calculation power, so they qualified as an ideal platform for the first revision of our runtime system.

2.2. Platform description

Independent of the micro controller or the operating system that is being used, nearly all possible platforms for ambient intelligence networks have similarities. This is due to the fact that these nodes have to remain active over a long period of time and have only limited energy resources [HD02]. The following, specific limitations are common for nearly all mobile nodes:

- Energy aware design:
  Since energy is the primary concern in ambient intelligence networks, all available platforms use micro-controllers of some sort. Apart from having only scarce resources in terms of memory or computation power, these micro controllers often also do not offer features that would be expected from a mainstream processor like separated addressing spaces, multi tasking or a memory management unit. Another similarity among all micro controller platforms is the very limited size of available data ram. Possible scheduling and memory management strategies have to be evaluated to ensure that SDL systems run reliable on these platforms.

- Communication hardware:
  Unlike pc platforms, nodes for embedded ambient intelligence networks usually provide a very low-level and time critical interface to the communication hardware. It must be evaluated which parts of the interface to the communication hardware have to be written in native code, and to which extend, the intelligence can be kept within the SDL system.

- Operating system:
  The operating systems used on these platforms are usually very basic, optimized for low memory and processing power consumption. Normally, they provide implementations of cooperative or even preemptive scheduling and message based communication. Time slices are not common in this domain, due to the unpredictability of task interruptions. The offered features of these operating systems range from simple runtime libraries to systems that allow multi tasking. Normally the operating system is directly linked against the application that is being executed, so there is only one application running on a ambient intelligence network node.

2.3. Available code generators

We did evaluate two possibilities further for transforming SDL systems into native code: The CMicro code generator that ships with Telelogic TAU [Tel] and a set of manual transformation rules for transforming SDL specifications into TinyOS compatible code [DRDK04]. Because Telelogic TAU is already integrated in our development process and because of the possibility of integrating the generated code into our ns+SDL simulator [KGGR05], we did choose the CMicro code generator for the first iteration. This code generator allows different integration types for integrating the generated SDL system into an operating system. To achieve first results, we did choose to use the bare integration offered by the CMicro compiler of Telelogic TAU. It provides a very loose coupling of the SDL system with the operating system. For the first revision of our runtime platform this was sufficient, because this integration methodology is very simple, as it only required a very loose coupling between the generated SDL System and the runtime platform. Although the CMicro code

generator does not support all SDL language features, like the priority input symbol, it provides a very special extension called signal priorities. This enables the developer to specify a global order that should be used when processing signals.

3. Requirements

We first identified the necessary functionalities and did a first test, whether integration with TinyOS could be successful. After the first promising results, we defined the requirements for the platform in terms of required services.

3.1.1. Functional requirements concerning hardware drivers

There are basically two methodologies for connecting hardware to SDL systems:

- Synchronous access by using blocking procedure calls – they block, depending on the used scheduler, at least the SDL process that issued the procedure call.
- Signal based access by sending signals to the environment. This is an asynchronous interface, because the signals are (at least virtually) processed concurrently to the running SDL system.

The following requirements have been defined, based on our experiences with the BicMon project and based on the requirements of a SDL specified media access control component:

- Compatibility to the SDL Environment Framework [FG05b] if possible. As a long term goal, all platforms should provide the same interface to the supported devices to SDL systems.
- Signal based access to the radio communication hardware. This should be a low level interface to facilitate the creation of MAC layers in SDL.
- Signal based access to the available serial (RS232) communication ports for communicating with local hardware like integrated sensor boards.
- Signal based and synchronous access to the available light emitting diodes for signaling and debugging purposes
- Signal based access to a unique ID that may either be provided by hardware, or that may be set during uploading the software on the microcontroller.
- Signal based access to the analog/digital converters
- Signal based access to the general purpose digital I/O pins

3.1.2. Additional functional requirements to the runtime platform

The following requirements were identified during our work on the BicMon demonstrator and are usable to a variety of possible SDL systems:

- Synchronous access to a random number generator
- Synchronous access to an interface for reading configurations
- Synchronous access to an interface for sending logging messages to the environment – probably by using one of the serial ports.

3.1.3. Non functional requirements to the runtime platform

Besides of the functional requirements, numerous non functional requirements have been defined for the SDL platform:

- Memory management paradigm:
  Dynamic memory should not been used, because of its unpredictability. It might be possible to allocate statically a small portion of the available memory for abstract data types or transmission buffers. This memory may not affect the other partitions, for example the SDL queue and the software platform must be able to handle out of memory situations correctly.

- Reliability issues:
  The platform must be reliable, so no unexpected behavior may occur. This is more deeply explained in Section 6.3.

- Portability:
  The platform should be portable to other SDL Kernels and code generators as well as to different hardware platforms. Also, the environmental interface that is offered to the SDL system should be as generic as possible. This should support a large range of different SDL systems without having to adapt the runtime platform.

4. Design

This section describes the main design rationale of our runtime platform. All relevant components are described as well as µSEnF, the SDL environment interface that is used to connect the generated SDL systems to the hardware.

4.1. Notation

The following notation is used for specifying the structure of the SDL runtime platform: Components with defined interfaces are connected by arrows. These interfaces are called interfacing points. There are mainly two types of interfacing points: Interfacing points that are connected with incoming arrows and interfacing points that are connected only to outgoing arrows. Interfacing points connected by incoming arrows are used by other components; these interfacing points are mapped to functions with defined names. These functions make up the exposed interface of a component. Interfacing points without incoming arrows are not called by external components – these interfacing points are normally represented by ordinary function calls in the source code.

4.2. Static structure

Because of the large number of possible configurations of a sensor node when considering the used SDL code generator, the used native operating system and the used hardware, a strict separation in independent layers with defined interfaces, called interfacing points became necessary to fulfill the portability requirement. Figure 1 illustrates the identified components and their relationships.

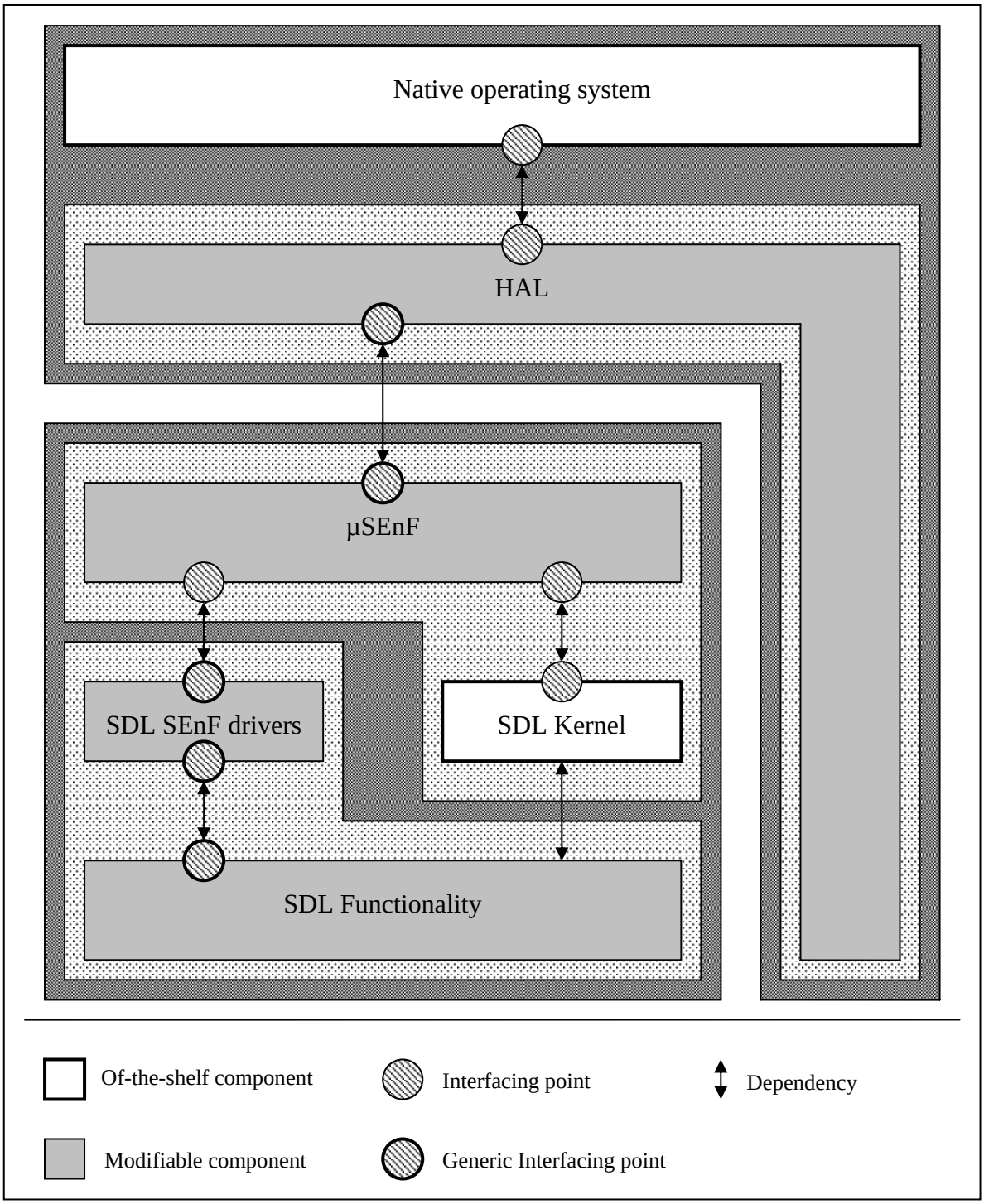

Figure 1: Static structure and dependencies of our SDL runtime platform

Figure 1 shows all components that are required for executing a SDL system on a resource limited platform as well as their dependencies and interfacing points. The components are drawn as squares while the dependencies are indicated by arrows, where the arrowheads show the direction of the dependency. Two types of components are distinguished in the diagram: Of the shelf components and components that can be modified.

Of the shelf components provide usually a more or less fixed interface. Two of these components have been identified: The SDL Kernel that depends on the used code generator or on the used transformation method and the native operating system that usually depends on the used hardware platform.

Modifiable components can be separated into those components that make up the runtime platforms, and into the SDL system that is to be executed. Of course, the runtime platform should be re-usable for multiple SDL systems, so the interface between SDL and its environment must be a generic one. The components that make up the runtime serve two purposes: They provide necessary functionality for executing these SDL systems, but they provide also glue code that maps the functionalities of the of-the-shelf components, that they depend on, to a more generic interface. As a result, two larger blocks can be identified:

- The generated SDL system
- The native interface

These blocks encapsulate their functionalities in generic interfaces; these are drawn as generic interfacing points – generic interfacing points mark blocks that can be exchanged. These are the native interface, the generated SDL system – and because the dependency of the SDL system with the used SDL Kernel is a special one, that is resolved by the compiler or the used transformation method, the SDL system itself. For example, the hardware abstraction layer can be kept when the used SDL Kernel – and as a result of this, also the implementation of the depending *µSEnF* is changed. This could be the case when moving from CMicro bare integration to CMicro tight integration. Depending on the similarities of the platforms, the code may partially be kept though. This makes this platform quite powerful, because it enables the developers to quickly bootstrap SDL systems on an existing platform, without having to worry about accessing platform specific hardware. The functionalities of these blocks is implemented in the components that it is made up of – these components are described in more detail in the following paragraph. They are separated into components that were created as a part of this work, into 3rd party components that were used out-of-the box and into those components that have to be specified by the protocol developer:

Created components:

- SDL SEnF drivers:
  These drivers make up the communication endpoint between the micro controller specific SDL Environment Interface *µSEnF* and the SDL system. They are implemented as SDL processes with defined names, which are addressed by the SDL environment when sending signals into the SDL system. They are also an abstraction layer between the SDL system and the concrete device that is used, allowing to quickly changing the used driver without having to rename signals in the SDL system. The interfacing points between the SDL SEnF drivers and the µSEnF are defined by every driver, for the SDL drivers this is a set of incoming and outgoing signals. The realization of the interface to C depends mainly on the used transformation method – and as a result, on the used SDL Kernel.

- The micro controller specific SDL Environment Interface (µSEnF):
  This component implements the environmental interface of the SDL system. This interface strongly depends on the used code generator - µSEnF in its current implementation is compatible to the CMicro code generator of Telelogic TAU. The µSEnF strongly depends on the SDL Kernel that has usually no standard interface – so µSEnF has to be re-implemented – at least partially – for every SDL Kernel that is to be used. In this component, also the methodology of sending signals into SDL is defined. Either the signals are stored at reception until they are polled, or they are sent asynchronously into the SDL system. The implementation of the interfacing points to the SDL drivers depend on the methodologies for sending signals to- and for receiving signals from SDL, that are offered by the SDL system. The interfacing points to the

hardware abstraction layer are realized as an ordinary interface of C functions – there is a set of functions defined for every device driver in the HAL for sending notifications, and also a set of callback functions in µSEnF for receiving notifications from the hardware. The µSEnF must also implement functions that are required by the generated SDL system to be able to execute – so the concrete implementation of µSEnF also depends on the used code generator.

- The Hardware Abstraction Layer:
  The Hardware Abstraction Layer serves as a defined interface between µSEnF and the hardware or the native operating system. While the interface to µSEnF is fixed, the interface to the native operating system strongly depends on the used operating system. Required functions that are not available though the operating system, have to be provided by the implementation of the HAL.

Components that have been used out-of-the box:

- SDL Kernel:
  This component relies on the used transformation method and is either part of the transformation framework, or of the used compiler. The coupling between this component and the SDL system is a very specific one, because this component provides the runtime library for SDL and the implementation of all SDL primitives. It also defines the interface that SDL uses to communicate with its environment, resulting in the need of a matching µSEnF component.

- The native operating system:
  Depending on the type of the native operating system, it might only provide some functions for accessing the hardware, or a more complex interface, providing also higher level operating system functionalities. The Hardware Abstraction Layer depends at least partially on the native operating system.

Components that have to be specified by the protocol developer:

- SDL Functionality:
  This is the SDL system that was specified by the protocol developer. It contains the functionality that is specific to this system as well as a set of SDL SEnF drivers. This SDL system is transformed by the used transformation method into native code. This component does not belong to the runtime platform being generated, but represents the functionality of the SDL system that is to be specified by the developer. The resulting SDL system contains this functionality, in any representation that the developer might choose.

The following sections give a more detailed overview on the components that were created as a part of this work.

4.3. SDL SEnF Driver

The SDL SEnF Drivers are a part of µSEnF. They use SDL signals to communicate with µSEnF by using the available input, and output functions for receiving signals from the environment and for sending signals to the environment. Mainly, they serve two purposes:

- Providing an abstraction layer between SDL and the real hardware
- Providing a communication endpoint

- Provide synchronous functionality

The abstraction layer provided by these drivers comes handy, when the used hardware is to be changed. For example, a set of SDL SEnF Drivers offer a generic serial interface to SDL, while to µSEnF, either specific signals for UART or Bluetooth communication are used. So the interface to SDL is specified by the offered functionality of the driver, in the example above serial communication, while the interface to µSEnF is specified by the concrete hardware that is used for providing this functionality. This enables the developer to quickly change the used hardware without having to worry about renaming signals in his SDL specification.

Another important role of the SEnF Drivers is that they might provide synchronous functionalities realized as SDL procedures. These are functionalities that are not mapped to environmental signals, but to direct calls into µSEnF using native Code. Since the offered methodology of integrating native C code into SDL may vary between the available code generators, parts of the SDL SEnF Drivers implementation might be required to be adapted to a different compiler.

4.4. SDL Environment Framework for micro controllers (µSEnF)

The µSEnF provides the interface between the SDL system and its environment. Although it was a goal to keep µSEnF compatible to the SDL Environment Framework SEnF that is used on platforms with more resources some changes became necessary. These changes are documented in Section 4.4.1 below.

The implementation on µSEnF depends on the used code generator, and on the used strategy for sending signals into the SDL system. We decided to asynchronously send incoming signals directly into the SDL system. Although this will not be a great benefit with respect to the achievable reaction time with our current bare integration, plans are to develop a runtime platform that is implemented by using a tight integration of the SDL system. This would allow us to directly affect the scheduling of the SDL system and to preempt SDL processes when necessary for achieving better response times.

4.4.1. Interface changes to SEnF

The data type used for representing data to be transmitted or received data was changed in µSEnF to respect the needs of resource constrained devices. The octet strings that are used by SEnF for sending or receiving data cannot be used with µSEnF for the following two reasons:

- Octet strings require dynamic memory allocation:
  The behavior of octet strings, if there is not enough memory available is not defined.

- Octet strings require deep copies of the signal data:
  If an octet string is to be sent to another process, a deep copy is made. Although this allows both processes to modify the data contained in the octet string, normally this is not necessary. Our experiences showed, that modifying a received packet, or packet that is to be sent, concurrently by multiple processes is rather uncommon.

These two issues prevented the use of octet strings for us. So we decided to introduce a new data type SEnFPacket that acts like a pointer type in C. Every instance of this packet type has a fixed size and is allocated from a ring buffer, so the software has to be aware of the possibility of having no packet buffer available when it is requested. Our plans for the near

future include supplying a set of SDL-patters that document the intended usage of this data type. Another necessary change was the representation of the time. On embedded devices, the time is usually measured in terms of ticks rather than using a real time clock. The duration of a tick – and therefore also the available timer resolution – strongly depends on the used hardware abstraction layer. A specific package has been added to µSEnF that cares about converting

4.4.2. Interfacing SDL with the environment

This section describes the current interfacing used for integrating SDL systems with the environment. Figure 2 shows the current structure of µSEnF.

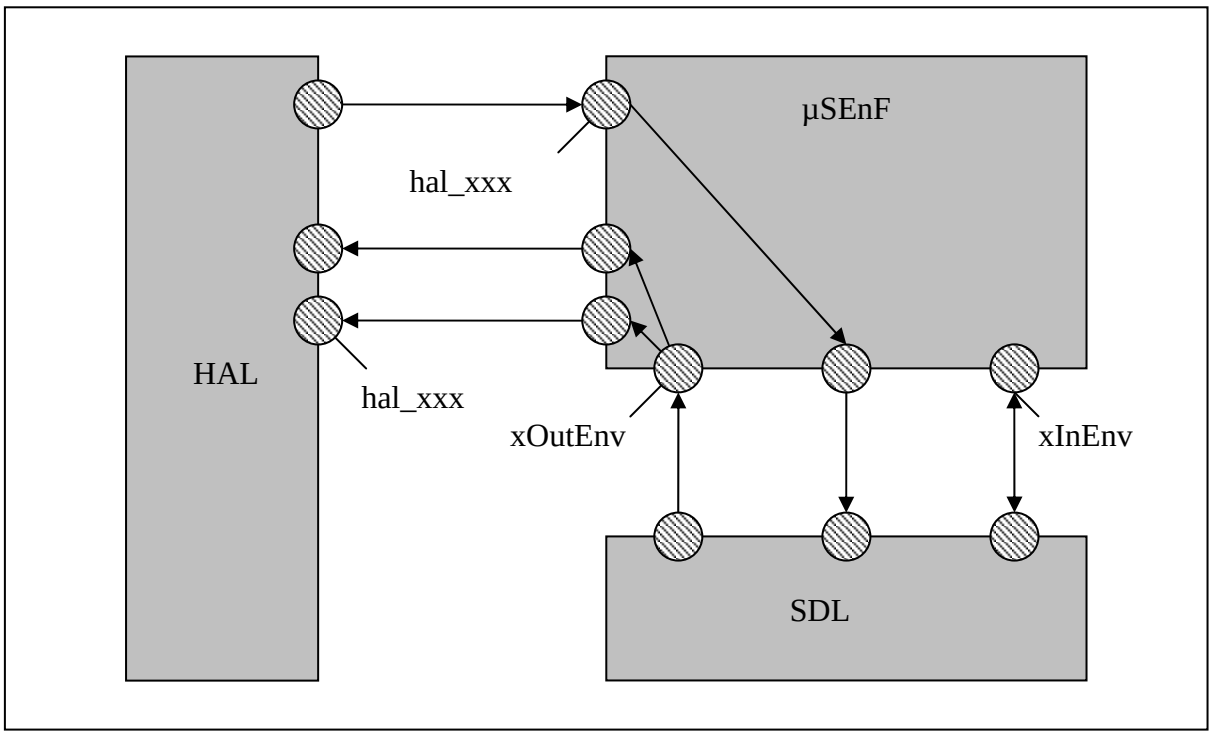

Figure 2: Structure of µSEnF

As indicated in Figure 2, several functions are used by µSEnF for communicating with the SDL system. The function xOutEnv manages the outgoing signals. Outgoing signals are directly mapped onto the Hardware Abstraction Layer, without any buffering. Callback functions are used to transmit incoming signals to the SDL system. The used code generator generates one queue that is shared by all SDL processes, and which contains all signals that are currently en route. If signal priorities are used, which is a proprietary extension of Telelogic TAU, the signals are sorted accordingly when they are inserted into the SDL queue. Afterwards, SDL starts processing the signal that is first in queue. Figure 3 illustrates this behavior.

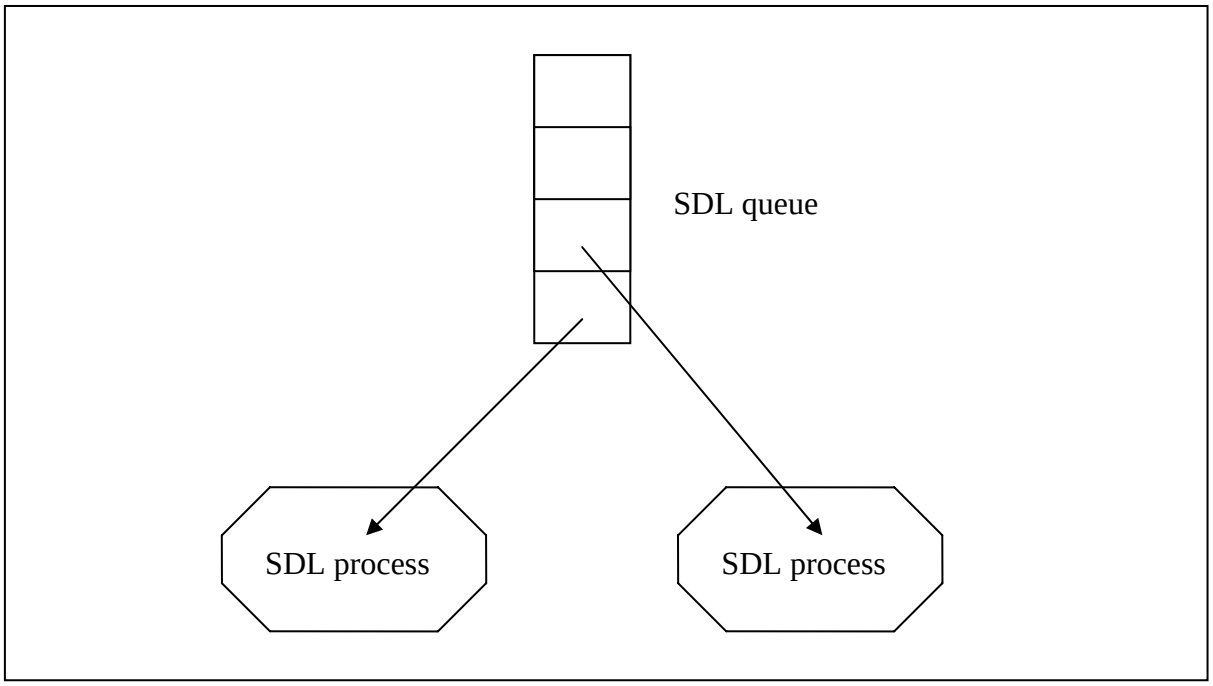

Figure 3: Queuing behavior of CMicro

As it can be seen, always the first signal of the signal queue is consumed. It can also be seen, that every signal already contains its receiver process. Due to this fact, every signal that is being sent into the SDL system from the environment must be addressed to a specific process. To keep the environment reusable, we decided to introduce the concept of the SDL Drivers that has been described above. Apart from providing functions for sending signals to the environment and receiving signals from the environment, µSEnF must also provide the interfacing points that are requested by the used SDL runtime. For the CMicro runtime using a bare integration, these interfacing points are described in Section 5.1.

4.4.3. Error handling

The µSEnF also contains error handling facilities. These error handling facilities usually catch unexpected situations, that can not be handled by SDL like signal drops due to full buffers or unspecified reception – although it is up to the view of the protocol designer whether this should be treated as error or not. Errors are reported by the supplied SDL Engine of Telelogic TAU. Currently, there are three strategies available to deal with errors: Ignoring them, this is suitable for expected and non critical errors like unspecified signal reception, to stop the whole system with an error message repeated to the serial port – this is especially useful while debugging or testing – or by issuing a complete reset of the network node to bring it back into a defined state – this is most common in production systems.

4.5. Hardware abstraction layer

The Hardware Abstraction Layer serves as a defined interface between µSEnF and the hardware or the native operating system. As indicated by the generic interfacing points, the interface to µSEnF is does not contain any operating system specific parts. The interface to the native operating system is not as generic and also strongly depends on the used operating system and its offered services. The HAL offers direct access to available devices on the micro controller like serial and radio communication. This component is an interface that maps the functionality of device drivers to the generic interface that is offered to the generated SDL system.

The generic interface between the Hardware Abstraction Layer and the generated SDL system is realized by a set of functions with specific naming conventions. Since this is a bidirectional interface, both blocks have to offer a defined set of functions to make the linking possible. Signals from SDL to the hardware abstraction layer are mapped by to µSEnF calls into the implementation of the HAL; signals from the hardware abstraction layer are sent to µSEnF by using the post functions that are provided by the generic interface of µSEnF. The concrete interface that has been specified for the generic interfacing points of µSEnF and the HAL is documented in Section 5.2.

5. Implementation

As a first result, we did implement our SDL runtime platform and our SDL Engine on a MicaZ mode running TinyOS. Since we did use TAU CMicro to generate C code out of the SDL specifications, the Implementation of the SDL Engine is specific to CMicro. The following sections briefly describe highlights of the implementations.

5.1. Bare integration of the TAU CMicro runtime

The bare integration of the CMicro code generator from the Telelogic TAU SDL Suite requires only very few integration with the used operating system. This section describes how a bare integration, the integration used by the first revision of our platform, can be performed.

5.1.1. Expected functions from the runtime system

The following service primitives are required to be present in µSEnF for performing a bare integration:

Timing related

These functions are used to interface the SDL system with the current system time.

- `xmk_NOW()`
  This function is called by SDL to request the current system time from the runtime environment.

- `xmk_InitSystime()`
  This function is called once when starting the system; it can be used to initialize the necessary hardware registers.

- `xmk_SetTime()`
  This function is called when the current system time should be set to a specific value. It might be called if the SDL system detects an overrun of the time.

Memory related

The following two functions must be present, even if no dynamic memory is being used by the SDL system due to the fact that the compiler cannot ensure that signal parameters will never grow too large for being handled by static memory allocation. Our runtime platform will issue an error, resulting in either a reset or a system stop if one of these functions is called. The used error handling depends on compile time settings.

- `xAlloc()`
  This function is used to allocate a block of a specified size.

- `xFree()`
  This function is used to free a memory block.

Input and output of signals

These functions are used by the SDL system for communicating with the environment.

- `xInEnv()`
  This function is called by SDL to poll signals from the environment.

- `xOutEnv()`
  This function is called every time a signal is sent from SDL into the environment.

- `xInitEnv()`
  This function is called once upon the start of the system. It can be used to perform required initializations.

Error Handling

This function is called by SDL when something unexpected happened, for example a dropped signal due to a full signal queue.

- `ErrorHandler()`
  This function is expected to handle an error. Depending on the actual error, it may be okay to return to the SDL system.

Handling of critical paths

These two macros have to be present to handle critical paths. They probably need to be integrated with the used native operating system.

- `XMK_BEGIN_CRITICAL_PATH`
  Start a critical path.

- `XMK_END_CRITICAL_PATH`
  End a critical path.

5.1.2. Offered services to the runtime system

The following main service primitives are offered for controlling the SDL system when using bare integration:

- `xmk_InitQueue()`
  This function must be called before any other provided function is being called. It initializes the SDL queue.

- `xmk_InitSDL()`

This function is to be called before xmk_RunSDL should be called. It initializes the whole SDL system.

- `xmk_RunSDL()`
  This function never returns. It runs the SDL system in an endless loop.

- `XMK_SEND_ENV()`
  This function can be used to asynchronously post events into a running SDL system or to send signals synchronously to SDL from within the xInEnv function when SDL polls for new signals.

5.2. Generic interfacing points between HAL and µSEnF

This Section documents the services that are offered by the Hardware Abstraction Layer, sorted by their categories, and the services that are required to be offered by µSEnF. Depending on the offered services of the micro controller, not all of these interfaces have to be present in a concrete HAL. However, interfaces must be implemented completely, so if, for example, radio communication is supported, all services that belong to radio communication have to be implemented.

5.2.1. Naming conventions for the interfacing points

Functions of the hardware dependant interface are currently prefixed with ‘driver_<device>’. Events from the hardware are signaled by using callback functions.

5.2.2. Radio communication

Currently, there is support for the ChipCon CC2420 Radio Transceiver chip included in the hardware abstraction layer. During the work on our QoS MAC layer MacZ [KF05], the necessary primitives for a low level interface from SDL to the hardware have been identified. In concrete, the following interfacing points have been implemented for sending signals to the transceiver chip:

- `driver_CC2420_Setup(Channel: Integer, TxPower: Integer)`
  This signal sets the channel that the transceiver chip will operate on as well as its transmission power for future transmissions. For the channel parameter the following constraint must hold: 10 < Channel < 27, for the TxPower parameter, the following must hold: 2 < TxPower < 32.

- `driver_CC2420_Mode(Mode: Integer)`
  This signal sets the CC2420 to one of three operating modes:

  - Mode 0: Power down mode
  - Mode 1: Idle mode
  - Mode 2: RX mode

  The TX mode is automatically set when a transmission request is sent to the driver.

- `driver_CC2420_Send(Data: void*, Length: Integer)`
  This sends the given data packet without checking the CCA pin prior to sending. Since pointers are not implemented in SDL, a special data type must be provided that encapsulates all necessary behavior.

- `driver_CC2420_SendCca(Data: void*, Length: Integer)`
  This sends the given data packet with checking the CCA pin prior to sending. Depending on the state of the CCA pin, either the packet will be sent or an error message is returned immediately.

The following interfacing points are expected by the HAL for signaling received events from the transceiver chip:

- `driver_CC2420_SFD(Success: Boolean)`
  This signal is either received as a response to a send operation or while the transceiver chip is in RX mode. When the transceiver chip is in RX mode, the reception of this signal indicates the receiving of a valid preamble and start-of-frame delimiter, indicating the arrival of a new packet. When this signal is received as response to a send operation, the value "true" for the parameter indicates the end of a successful transmission – the value "false" is only returned as answer to CC2420_SEND_CCA and indicates a busy medium. Note that a value of true is not equivalent to a successful reception of the signal at any receiver node.

- `driver_CC2420_Sending(Success: Boolean)`
  This signal is always sent to the SDL system as a response to CC2420_SEND, or as a response to CC2420_SEND_CCA if the medium was idle. It indicates the successful beginning of a send operation. The parameter is currently unused.

- `driver_CC2420_CCA(Idle: Boolean)`
  This function is called for reporting the state of the medium. This signal is only issued when the state of the medium has changed or the runtime is unsure about the current media state – for example after transmitting some data. If Idle is true, then the medium is idle, otherwise it is busy.

- `driver_CC2420_Recv(Data: void*, Length: Integer, GoodCRC: Boolean, RxStrength: Integer)`
  This signal indicates a completely received packet. The first two parameters must be encapsulated for SDL because pointer handling should be omitted. The third parameter indicates whether the CRC check was successful, the forth parameter indicate the strength of the received signal.

5.2.3. Serial communication

The following interfacing points are available to transmit signals from SDL system to the hardware abstraction layer.

- `driver_UART_Send(void * data, int length)`
  This function sends a packet to the serial communication port with the number 1.

- `driver_UART_xSend(int port, void *data, int length)`
  This function sends a data packet to the given serial port. The available port numbers start with 1.

- `driver_UART_Setup(int port, int baudRate)`
  Calling this function sets up the serial communication for the given port number.

The following interfacing points are expected by the Hardware Abstraction Layer to signal events to the SDL system.

- `driver_UART_Recv(int port, uint8 octet)`
  This signal indicates the reception of an octet via the serial communication interface. Please note that this interface does not preserve message boundaries. The first parameter contains the port where the data was received from; the second parameter contains the received octet.

5.2.4. Light emitting diodes

The following interfacing point is available for controlling the light emitting diodes of the micro controller.

- `driver_LED_Set(int operation, int bitmask)`
  This function changes the state of the available light emitting diodes by applying a logical operation and a bit mask to the current state of the lamps. The available operations are currently set, and, or and xor.

5.2.5. Access to unique ID

This interface has not been specified yet and is considered to be future work.

5.2.6. Access to the analog/digital converters

This interface has not been specified yet and is considered to be future work.

5.2.7. Access to the digital general purpose I/O

This interface has not been specified yet and is considered to be future work.

6. Evaluation

This section evaluates our current platform. The results presented here are preliminary, because the development of the runtime platform is still work in progress. Two criteria for the evaluation are separated: Timing issues and reliability issues. For being able to fully understand these issues, some specifics about SDL must be known. These specifics are described in the following section.

6.1. SDL Model

SDL assumes a theoretical model of communicating extended finite state machines that have infinite queues for storing incoming signals. In the model of SDL-96, these extended finite state machines are running concurrently. A running state machine executes a transition whenever it is triggered. Transitions may be triggered by the following events:

- Received signals
- Continuous signals
- Spontaneous transitions
- SDL timers

Received signals are signals that have been sent to a specific process, either by another SDL process or by the environment. Upon reception of a signal, the associated transition is executed.

Continuous signals cause a transition to fire every time the signaling queue of a process is empty. This construct should be avoided, because the order of scheduling SDL processes is not specified – so, depending on the used SDL Kernel it might happen that the process with the enabling condition will be the only running process. Since the scheduling order of the processes is not defined in the SDL standard [SDL100], it is not possible to avoid this case – whether this could happen or not depends on the used runtime environment. Therefore, this construct should be avoided when developing embedded systems – in the further work, continuous signals will not be considered when doing formal verifications.

Spontaneous transitions may fire at any possible time. This is used to model behavior that has not been implemented or to specify indeterminism, for example when a medium with the ability to loose messages should be specified in SDL. In production systems, this construct should be omitted, so it will not be considered further.

SDL timers may cause a transition to fire at a specific time. Unfortunately, only the minimum waiting time for a timer can be specified – it is up to the timer, and probably also depends on the implementation, whether it fires at this point of time or at any later time.

Since all state machines are running concurrently in the theoretical model, they may also concurrently fire transitions and process signals or timers. When this behavior is mapped to a real hardware, two mappings are possible:

- Preemptive scheduling
- Non-preemptive scheduling

When a preemptive scheduling is implemented, processes may interrupt each other when they receive a signal. In Telelogic TAU, this is combined with a priority mechanism, so that only lower priority processes will be interrupted.

One interesting extension of Telelogic TAU is the ability to define signal priorities. Unlike the process priorities that have been mentioned above together with preemption, signal priorities may be used to associate signals with a specific priority. So the signals with higher priority will be inserted in the SDL queue before the signals with lower priority. This guarantees that higher priority signals will be processed before the lower priority signals.

The following section will evaluate the timing constraints that hold for the current non-preemptive implementation of our SDL runtime.

6.2. Timing

Two main areas can be spotted when considering timing issues in the embedded systems domain:

- Required time for reacting on events
- Timer accuracy

For being able to predict the required time for reacting on events, it is assumed that signal priorities or a similar extension is being used, ensuring that the measured timer signal will

receiver the highest priority. Further work will also consider the achievable timing accuracy for lower priority signals.

6.2.1. Reaction time on events

The concrete timing heavily depends on the used runtime platform, and on the used SDL kernel. The used hardware platform affects the timing by its timer accuracy and by the processing power of the used processor. A kernel that supports preemption can interrupt processes for higher priority signals, while a kernel that does not support this feature has to complete the execution of the currently running transition before the newly arrived event can be handled. So preemption usually results in a better response time to high priority signals. For being able to start processing an event after it arrived in the SDL system, the currently running transition must be completed and the new process has to be scheduled. For the case that that newly arrived signal has the highest possible priority, the following sources for relevant delay have been identified so far:

- Time required for passing the event from hardware to SDL
- Scheduling and dispatching of transitions
- Execution time of transitions

The following measurements show the time that is required for processing a signal from another SDL process, and the required time for processing a signal that was sent into the SDL system by the environment.

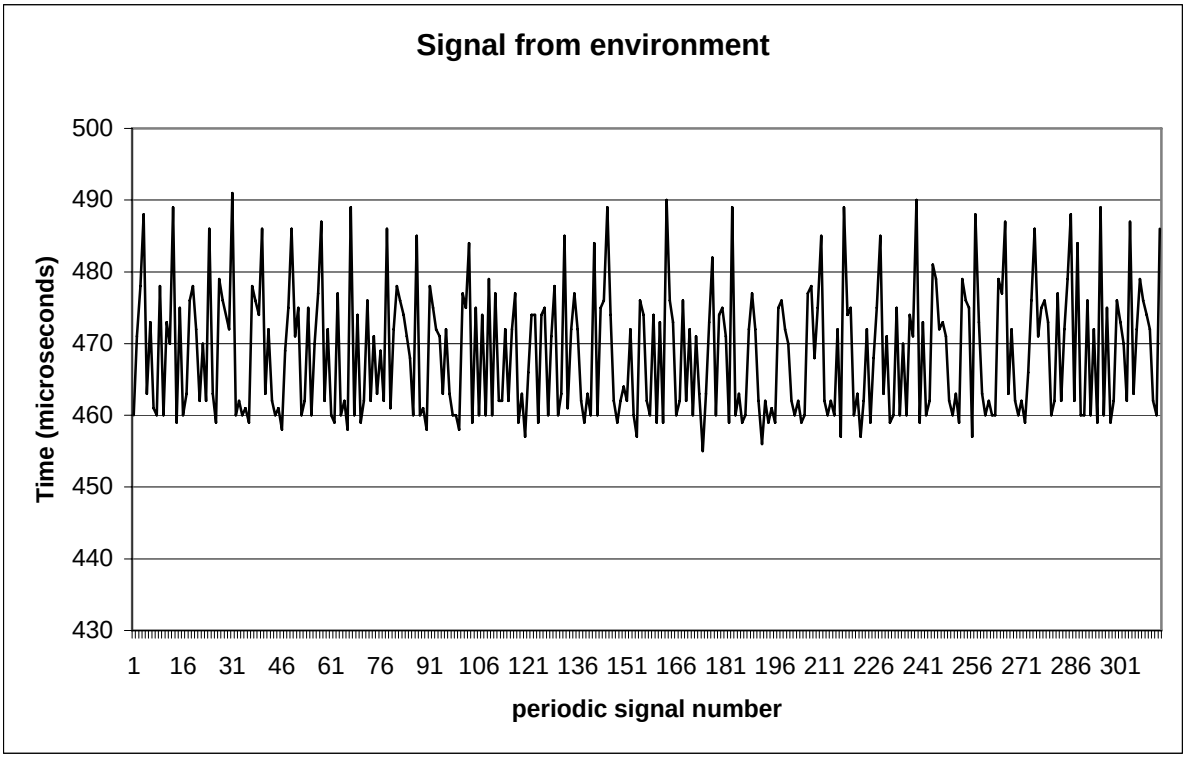

Figure 4: Reaction time for signals from the Environment

As it is shown by Figure 4, a SDL process requires an average time of 470 microseconds to react on a signal from the environment. The measurement was performed with a SDL system that is almost idle. The next figure shows the measurement results for signals from another SDL process.

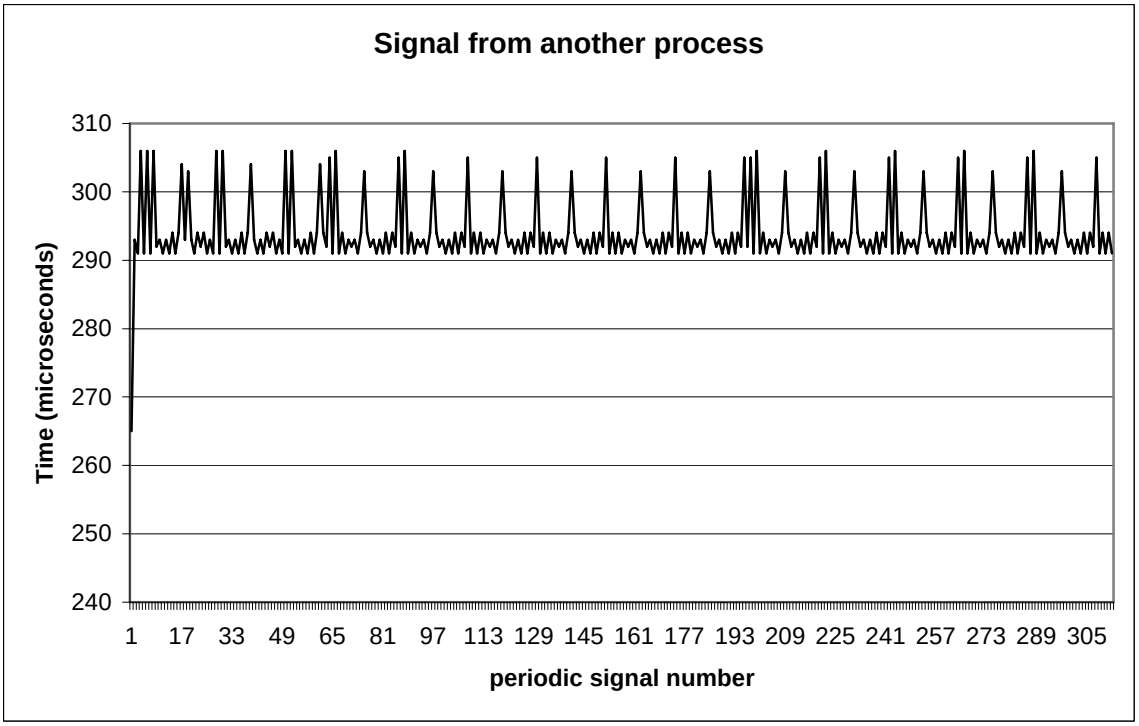

Figure 5: Reaction time for signals from the Environment

Figure 5 shows, that our SDL platform is faster when reacting to an internal event, but also that there is still a considerable delay and a jitter that is in the range of 20 microseconds. The measurement results show, that there is still room for improving the timing performance of our SDL runtime platform on MicaZ nodes.

6.2.2. Timer accuracy

Although SDL timers can currently not be set to fire within a defined period of time, the implementation usually does its best to fire the timers as accurate as possible. Although this is not yet sufficient for any real time scheduling, it might be a starting point. Since no preemption is currently used, the achievable timer accuracy heavily depends on the load of the SDL system. Improving the achievable timer accuracy is currently ongoing work.

6.3. Reliability

The reliability of a running SDL system is affected by the mapping from the theoretical SDL runtime model to a concrete platform. This is especially true for the mapping of the theoretically infinity sized signal queue to a size limited real queue. In the SDL model, every process has a signal queue that holds all signals addressed to him, that are waiting to be processed. Due to the limited size of these queues in reality, the problem of possibly full queues must be addressed. This problem becomes even more evident, if the amount of available memory is very small. To overcome the problem of a full message queue, the causes of this problem must be addressed, as well as possible methods for handling such an error must be evaluated.

6.3.1. Causes for full queues

The SDL signal queue has only a fixed size, which is in embedded devices usually relatively small. To understand the problem of full queues, the SDL model must be understood. In SDL, nearly every transition is triggered by an incoming signal or by a timer. The only exceptions

to this rule are continuous signals and spontaneous transitions, which have been described above. Since both constructs should not be used in embedded systems, these are not considered further. When not considering these two exceptions, there are three possible situations, where signals might be added to the SDL queue, and possibly cause an overflow:

- A running transition is sending a signal
- A timer expired
- A signal is sent from the environment to SDL

6.3.2. Possible solution

These situations have to be investigated more deeply in future. For coping the first situation an understanding of the SDL system is necessary, at least to a certain extend. Projections could be used to abstract from the functional behavior, by just considering the possibly sent and received signals by every process. With a more simplified view on the SDL system, possibly a methodology could be created for analyzing the behavior of the SDL system with respect to the signals that are created and consumed by its processes. It could eventually be proofed, that the maximum number of signals would not extend a certain value. This would be the maximum required queue size.

Timing issues must also be considered when starting to create a model for determining the required size of the SDL queue. When a timer is set, it will fire a signal to the receiver process at the time that it expires or any timer later, but by no means any earlier. In the meantime, the SDL system is able to fire a certain amount of transitions. This can either reduce or increase the amount of signals in the SDL queue – so the worst case must be expected. Since every SDL timer can be in the SDL queue only once, according to the SDL standard [SDL100], it should be possible to model the amount of required signaling space. Eventually, projections can be used to identify timers that are never concurrent to each other, these timers could share one requested queue space then.

Another issue that must be considered is the reception of signals from the environment. Since these signals can be completely indeterministic, we propose to extend the SDL semantics for environmental signals. A possible solution could be to limit the number of environmental signals that can be in the SDL queue. This limitation could be a global one, or it could be grouped by the type of signals – for example for one type of signals the most recent could be kept when the maximum amount of signals is reached, for another type the first signals could be kept while the newer ones would be dropped. Creating a framework for ensuring that queues do not overrun is current work in progress.

7. Conclusions

Our first platform did proof that it is possible to instantiate a model driven development process on resource limited platforms, like the MicaZ nodes. The developed runtime platform has a clear structure, with defined, extensible and light weight interfaces which encourages portability either to a different SDL Kernel or to a different hardware platform. The timing issues are still a problem that is worked on, since the model of the traditional SDL timers and SDL signals, maybe a modification is necessary to introduce deadlines – this would enable the SDL model to become capable of real-time processing. First effort in this direction was already taken, but further research in this area is necessary. This also holds for the reliability issues.

8. Further work

Since this is a report on work in progress, numerous work packages still remain open. The following paragraph lists the works that are currently ongoing or planned for the near future.

The first issue is the abstract data types and their generators of SDL. These data types must be implemented on these resource limited platforms in a very efficient manner. Also a methodology should be implemented, to notify the SDL system when there is no more space left for storing more data.

Another important point is the reliability of SDL specifications. This is an issue that requires further research. The possibility of creating a formal, static technique for analyzing a SDL system with respect to the possibility of queue overflows should be evaluated and such a technique eventually developed.

Also the timing issues must be further investigated in future. Maybe the SDL scheduler can be adapted to support deadlines. [Kol00] did already integrate an earliest deadline first scheduling into a SDL scheduler.

The possibilities of a tight integration, and also the pros and contras of using a preemption-enabled kernel should be evaluated. The use of the SEnFPacket type should be documented, probably by specifying a micro protocol [FGGS05].

9. Definition of important terms

This section lists and defines important terms that are used in this report.

SDL System

The SDL System is specified by the developer. It contains the functionality of the developed system.

SDL Kernel

The SDL Kernel ships with the used code generator or with the used transformation framework. The services that are contained in the SDL kernel may vary – together with the SDL environment all necessary functions must be provided to execute the generated or transformed SDL system.

SDL Environment

The SDL environment provides the connection of the SDL system to real hardware, or to other processes as well as necessary services, that are not implemented in the SDL kernel.

SDL Engine

The SDL Engine is composed of the SDL Kernel and the SDL Environment and contains all services that are necessary to execute a SDL system that has been generated with a specific transformation method.

Generated SDL System

The generated SDL system is the SDL system that has been transformed by a transformation method into native code and linked with the SDL Engine to form an executable.

10. References

[DRDK04] D. Dietterle, J. Ryman, K. Dombrowski, R. Kraemer. *Mapping of High-Level SDL Models to Efficient Implementations for TinyOS*. Euromicro Symposium on Digital System Design (DSD'04), pp. 402-406, 2004.

[DSH99] M. Doerfel, F. Slomka, R. Hofmann. *A Scalable Hardware Library for the Rapid Prototyping of SDL Specifications.* Tenth IEEE International Workshop on Rapid System Prototyping (RSP'99), 1999.

[DZM01] C. Drosos, M. Zayadine, D. Metafas. *Real-Time Communication Protocol Development Using SDL for an Embedded System On Chip Based on ARM Microcontroller*. 13th Euromicro Conference on Real-Time Systems (ECRTS'01), 2001.

[ADLPT99] J. M. Alvarez, M. Diaz, L. Llopis, E. Pimentel, J.M. Troya. *Embedded Real-Time Systems Development Using SDL*. In proceedings of IEEE Real-Time System Symposium WIP sessions. RTSS'99, 1999.

[HSWHCP00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. *System architecture directions for network sensors.* ASPLOS 2000, Cambridge, November 2000.

[WT04] T. Wroldsen, S. Tveitane. *A Real Time Operating System for embedded platforms.* Masters Thesis at Agder University College in Norway.

[FGGS05] I. Fliege, A. Geraldy, R. Gotzhein, P. Schaible: *A Flexible Micro Protocol Framework*. In D. Amyot, A.W. Williams (Eds.), System Modeling and Analysis, Lecture Notes in Computer Science 3319, Springer, 2005, pp. 224-236.

[KGGR05] T. Kuhn, A. Geraldy, R. Gotzhein, F. Rothländer. *ns+SDL - The Network Simulator for SDL Systems.* 12. SDL Forum, Grimstad, Norway. June 2005.

[HD02] J. Hill and D. Culler. *Mica: a wireless platform for deeply embedded networks.* IEEE Micro, 22(6): 12-24, November/December 2002.

[AHS02] E. Aarts, R. Harwig, M. Schuurmans. *Ambient intelligence*. The invisible future: the seamless integration of technology into everyday life, Mc-Graw Hill, 2002.

[Kol00] Thomas Kolloch. *Code generation with SDL -- an integration with RTEMS and message deadlines.* Technical report TR-SDL00, Lehrstuhl für Realzeit-Computersysteme, Technische Universität München, München,

Germany, January 2000.

[FG05a] I. Fliege, A. Geraldy, R. Gotzhein, T. Kuhn, C. Webel. *Model-Driven Engineering of Ambient Intelligence Systems with SDL: Design, Implementation and Performance Simulation.* Technical report 342/05, University of Kaiserslautern, 2005.

[FG05b] I. Fliege, A. Geraldy, R. Gotzhein, T. Kuhn, C. Webel, C. Weber. *Konzept und Struktur des SDL Environment Frameworks.* Technical report 341/05, University of Kaiserslautern, 2005.

[KF05] T. Kuhn, I. Fliege. *Micro-protocol based design of a highly adaptive and integrated QoS MAC layer for Ambient Intelligence Systems.* Technical report 347/05, University of Kaiserslautern, 2005.

[Cin] Cinderella. *http://www.cinderella.dk*

[Tel] Telelogic. *http://www.telelogic.com*

[Pra] Pragmadev. *http//www.pragmadev.com*

[SDLRT] SDL-RT standard. *http://www.sdl-rt.org/standard/V2.1/pdf/SDL-RT.pdf*

[SDL100] SDL standard. *http://www.sdl-forum.org/Publications/Standards.htm*