A Real Time Operating System for embedded platforms

by

Torstein Wroldsen
Ståle Tveitane

Masters Thesis in
Information and Communication Technology

Agder University College
Faculty of Engineering and Science

Grimstad, May 2004

Summary

SDL (Specification and Description Language) is today widely used for description and development of complex systems. One of the major benefits of SDL is the possibility to graphically describe a complex system, as well as the capability to analyze the system before implementation. This thesis evaluates SDL as a formal description language for use in an embedded platform.
To be able to map the properties and behaviour of an SDL system into a programming language, a Real Time Operating System (RTOS) must be used. We will evaluate all properties of SDL to get a overview of which properties can be mapped to a programming language, which properties can be omitted, and which properties isn't possible to map to a programming language.
Further, well evaluate what properties of an RTOS are essentially for the implementation of an SDL system.
Several RTOS already exists on the market today. Some are specifically developed with mapping between SDL and ANSI C as the main task.
Our target is the Atmel AVR 8 bit microcontroller. A microcontroller built on the Harvard architecture (separate memories and buses for program and data).

Through this master thesis we have discovered the shortcomings of several RTOS. Due to this, we have developed our own RTOS, SDL REFLEX, specifically designed to our needs and with the AVR microcontroller as target.

Preface

This thesis was written for Agder University College, Faculty of Engineering and Science and it is a part of a "Norwegian master grad". The work has been carried out in the period January 2004 and May 2004.

We would like to thank our supervisor, Paul Bjørn Andersen at Agder Univeristy College for valuable help and inspiration.

Table of content

1. Introduction

1.1 Background

From the early stage of the computers until now, software designers have experienced a tremendous acceleration regarding development time and language complexity for the different programming language.
In today's competitive development environment, organizations are struggling to deliver more complex systems in less time and with fewer staff. For developers this has serious consequences; it increases their responsibility to design and deliver the highest quality systems as efficient as possible. In this climate, many developers find that adopting a more visual, automated and reliable development process – model-driven development – can help.
With the rising complexity, and the importance of rapid development, designers tend to use software automation tools. However, design automation in general needs a formal system description to capture the functional and non-functional requirements. Model-based code generation produces application source code automatically from graphical models of system behaviour or architecture. Development tools are moving to model-based development to raise the level of abstraction at which the developers can work.
The fast-paced and competitive world of embedded systems technology forces manufactures to reduce time to market. Object-oriented (OO) methods and tools can improve productivity, quality, and reuse. Capabilities for generating code from object models offer additional help for keeping pace.
OO methods help developers analyze and understand a system, but the bottleneck of analyzes and design has been the transition to code. Without the automatic code generation, the benefit of object modeling seldom live thought the entire products life cycle. Developers pressed on time to marked, or deadlines for upgrade as well, tend to make changes directly to the source code, resulting in out of date models. Generating code from object models still retains it usefulness.
Model-based code generation continues a long term trend in development tools. The abstraction of the languages has increased from assembly to high-level languages up to graphical models. The abstraction has moved from the system solution space toward the application problem space. The reason for this trend is the demand for increased productivity, the need for constructing larger applications and last but not least understanding complex systems. The enduring nature of these needs suggest that model based code generation is inevitable.

1.2 Thesis Definition

The thesis is closely related to the current research and development of embedded platforms at HiA, Grimstad. The use of formal description languages for development and documentation of applications is one of the main focuses for using fast prototyping software. At present, real time operating systems for description of state machine behaviour and state transition exist for several different embedded systems.
The title for this thesis is "A Real Time Operating System for embedded platforms".

The final definition of the project is:

- Analyze which properties a Real Time Operating System (RTOS), should have to be well suited for implementing systems described in a formal language such as SDL on an embedded platform.
- Explore commonly used RTOS, and analyze their properties.
- Design and implement an RTOS for the Atmel AVR microcontroller family with properties as found in the above research.
- If time permits, use this RTOS in a test case on HiA's recently developed Lego™ training kit.

2. Formal Description language

Today the developers of embedded systems are exposed to a fast-paced competitive arena. The result of this exposure is a demand for reduced time-to market. One way of achieving better and faster results is to use formal description languages. Several different languages exist such as Unified Modeling Language (UML), Model Driven Architecture (MDA), Component Based Development (CBD), Use Case Maps, Message Sequence Chart (MSCs) and Specification and Description Language (SDL). The common thing with all of them is that they support modeling concepts.
UML is today de-facto standard for specification, construction and documentation of software systems. UML is an open standard developed by Object Management Group (OMG). In general UML is a language for constructing framework and classes for a system.
SDL originated from the telecom industry as a tool for unambiguous specification and description of the behaviour of telecommunications systems. Every concept and construct in SDL has a precise meaning and can be executed.
Some of the benefits using a description language include:

- System is divided into modules. Gives easier overview of a complex system.
- The possibility to analyze a module or several modules in the system before implementation.
- Testing interaction within the system can be done before implementation.

2.1 Object models

Object-oriented (OO) methods and tools can improve productivity, quality, and reuse. Automatic code generation based on graphical models is one of the approaches used to achieve the above mentioned demands. Model-based code generation produces application source code automatically from graphical models of system behaviour or architecture. Development tools are moving towards model-based development to raise the level of abstraction at which developers can work.

2.2 Basic approaches: structural, behavioural and translative.

These three approaches seem to cover the today's available methods and tools. As illustrated in figure 1, the "next" approach incorporates the properties of the previous approach. The first approach, structural approach, generates the framework code for the object structure of a system. Behavioural approach model the behaviour of a system sufficiently to enable generation of code for the system functionality. The most recent approach also adds an architecture model that enables user control of all generated code.
All the code generating model-based tools have some in common, such as:

- Regardless of which approach is used, they are all associated with at least one method of analysis and design
- Commercial tools on the market support the approaches.
- All translation tools take as input the models of a system's static object structure, which are developed using the OO method supported by the tool
- All tools translate these models into corresponding code for system objects, providing a framework to implement for communication and behaviour code
- The synchronization of code and model are supported by most all tools.
- State machines are the common way of describing behaviour, whether or not code generation is supported.
- Real-time embedded applications have been developed by all approaches.

There are also some differences in the properties of the approaches. These differences might include:

- OO Methods associated.
- Verification of behaviour before code generation.
- Programming languages supported (other than C++).
- The amount of code generated (does it include behavioural code?).
- Synchronizing or reconciling of models and code.
- The possibility to customize the translator technology.
- Control of generated code and system architecture.
- Integration of non-generated code.

We will walk trough the three different methods, one by one. As a guideline one can state that structural approach generate code frames as a result from the static relationship among objects (UML), such as classes and relationships between classes. The behavioural approach uses additional state machines models and state transitions description to generate code for a whole system (SDL). The last approach is based on the independence between the architecture model and the application model. This approach leaves the programmer with total control over translating complete models into code.

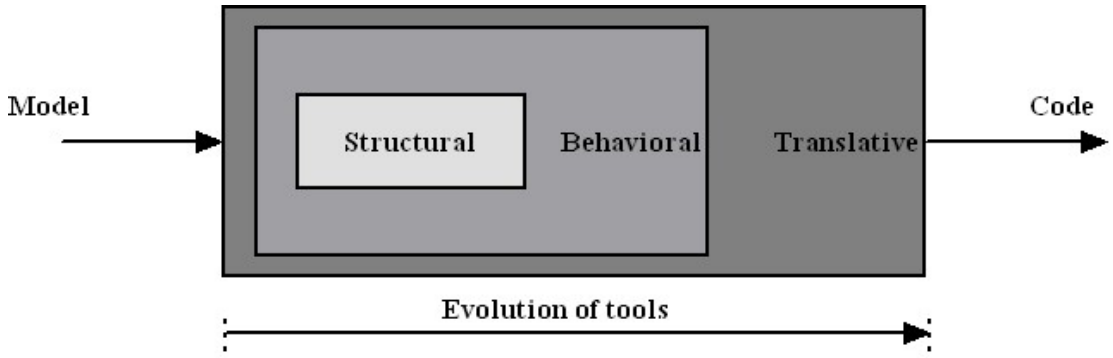

Figure 1.

As shown in figure 1, each can be seen as building on the concepts of the prior approach. The "inner" approach generates the framework code for the object structure of a system. Newer approaches model behaviour sufficiently to enable generation of system functionality. The most recent approach adds an architecture model that enables user control of all generated code.

2.2.1 Structural approach.

The basis for the structural approach is the model of object structure (static relationships). Based on this model most OO tools have the capability to generate code for the application framework. The code generated would normally be a class hierarchy, e.g. C++ or Java class hierarchy. The programmer can then add code for the behaviour of the different objects and the communication between them. This is done in the target language, being Java or C++ as some examples. The most common method of development using this approach is a gradual transition from analysis model to design and code. Methods that support structural code generation include Object Modeling Technique (James Rumbaugh), Object-Oriented Software Engineering (Ivar Jacobson), Object-Oriented Analysis and Design (Grady Booch). These methods joint together is what we today know as UML. The recent version out is UML 2.0.
As the name indicates, the approach doesn't support code generation for object behaviour. Methods in this category model behaviour as state machines without executable semantics. Developers then have to manually program the system dynamics of the application. The tools normally integrate hand code with

generated code. Some of the tools also protect the hand code from being overwritten if a regeneration of the object is done. If using tools that do not support such feature, the programmers have the tendency to develop two independent versions, the graphical model and the coded model. Some tools also support reversed engineering of OO programming languages, meaning that object structures can be generated from source code. Structural code generation is incomplete, but saves hand coding and provides an implementation framework consistent with the object model.

2.2.2 Behavioural approach.

This approach is based on state machines behaviour with action specification. Some OO methods that model behaviour with state machine, add code to represent action that occur upon a state transition. Tight coupled with objects of model structures as well as communication, the possibility for code generation is present. Code can be generated for the entire application.
Those tools that support behavioural system description, as the case with SDL tools, can simulate and test the model before code is generated. The application can be debugged on a graphical level of abstraction. SDL is only one of several description languages that support the behavioural approach. Most recently, UML 2.0 has implemented SDL, so UML 2.0 also has a part that is classified as behaviour approach. Since ULM 2.0 is the leading standard for formal description language, an introduction to UML will be presented in chapter 2.3. When modeling the behaviour of an entire system, the classical state-machine is described in two different ways. Methods using this approach also differ in the way they model object structure and communication. SDL includes block diagrams, process diagrams, signal and data type definitions and message sequence chart for object communication. Most SDL tools maps processes to an RTOS task, signal to messages in memory and timer to a hardware or software timer device.
The most significant difference in the approaches is that the in the latter system behaviour coding is done during object modeling, reducing the hand coding to a minimum. Another difference is the “lack” of reversed engineering, obvious since all code comes from the models. To be able to describe a system with behavioural approach the developer must obtain a state-machine view of the systems functionality as well as an object view of the systems structure.

2.2.3 Translative approach.

Translative approach is based on application and architecture models. The two models are totally independent of each other. Object-Oriented Analysis (OOA) is used to create an application model consisting of object structure, behaviour and communication. Also this approach has the ability to simulate system behaviour before code generation, just like the behavioural approach. The architecture model defines the mapping rules for the translation engine when generating code of the application model. One of the biggest benefits of this approach is the possibility to easy reuse code. The reason for this is the total independence of the application and architecture models.
To be able to simulate before code generation, the system is divided into domains. The domain is modeled by an object information model, state models for each objects and action specification for each state. Completely modeled domains can then be tested and if desired code can be generated. The simulation is based on an interpretation of the action specification, modeled in the domain.
As mentioned, architecture model consists of the translation rules for mapping of the OOA construct into source code. All constructs, if code is to be generated, used in OOA must have a defined translation in the mapping. The developer has the benefit of complete control over the code generation, since he also controls the development of the mapping rules in the architecture model. Programming language can also be changed to meet the needs of the programmer. Performance critical parts can be translated directly into assembly for best performance control. Another possibility is to omit mappings for some implementations in the application model. The developer then have to hand code the unmapped part, letting the developer have the total control of the quality of the code.
The translative approach do not support reversed engineering, if the system need to be changed this must be done in the application model or the architecture model, and regeneration of code must be done for applying changes.

2.2.4 Comparison between the approaches

This is a simplified comparison for the three models described above.

Approach	Structural	Behavioural	Translative
Models Supported:	Objects	Objects / states / actions	Objects / states / actions/ architecture
Methodology:	OMT, Booch, UML, …	Harel, SDL, ROOM, UML 2.0	Shlaer-Mellor (OOA)
Languages targeted:	C++, Ada, …	C++, C, …	Any (user-defined)
Extend of code generated:	Framework	Complete	Full, any architecture
Control of code:	Templates	User code, RT libraries	Complete, separate, architecture model
Tool support:	Reverse engineering protected code	Simulation and debug	Simulation and debug, construct architecture
Cost:	Small	Tooling, learning	Building architecture
Advantages:	Synch code to model	Early verification	Reuse architecture

Table 1. Comparison between different approaches.

2.3 UML, a short description

UML is a visual language for specifying, constructing and documenting software systems. UML is an open standard and has established itself as a common modeling language throughout the software and system industry, and is treated as today's de facto standard for specifying software architecture. OMG is the body responsible for the standardization of UML, which started with UML 1.1 in 1997. During this project UML 2.0 was the current standard. March 2003, UML 1.5 was released, but had only small changes from version 1.4. The UML 2.0 upgrading is due to the fact that version 1.x has lots of shortcomings (e.g. non-standard implementation, exaggerated complexity, imprecise semantics etc). One inadequate part of UML 1.x often mentioned is the insufficient support for component-based development. To cope with these drawbacks UML 1.x has been used in a non-standard way. Other tools like SDL, ROOM10 and MSC11 have been used together with UML 1.x to achieve the goal.
The basic improvements from version 1.x to 2.0 are:

- Better support for component-based software development.
- Better alignment of other widely used standards like, XML/XMI and SDL.
- Improved support for composite state machines.

Beginning from UML 2.0, SDL 2000 is implemented as a standard for describing system in the state-machine behaviour approach. The classic UML diagrams, from UML 1.x, are used for describing the overall framework of the application. SDL is then used to describe the system interaction and transitions in a process. Since the classic UML part is out of scoop for this project, SDL will be the main topic and UML will be left from this point.

2.4 SDL - The Nature of Real Time Applications

SDL is a formal description language defined by the ITU Z.100 recommendation. SDL today is a language well suited for specification and implementation of distributed systems. SDL was developed by the telecommunication industry. At the early start in 1972, SDL was originally a language for specification. As a result of the first tool being released in 1984, things changed significantly. Both the user and the designer of SDL had to be more formal. The workload then became more apparent, but the benefits were the identification of errors and the ability to animate models, so "what if" questions could be answered.
In 1992 SDL got some significant updates, the addition of type constructs for an object oriented version of SDL. This version of SDL is called SDL-92.
After the release of UML 2.0 there has been a merge between UML 2.0 and SDL-2000, which is the latest SDL recommendation.
UML 2.0 has built in support for all of the SDL semantic.

UML will be left as a subject, and the focus will fall on SDL, in particularly SDL-2000.
SDL is broadly used in the telecommunication domain, especially on protocol specifications, as well as on the specification of reactive real-time and embedded systems. Since SDL is a formal language, it is widely used for specification, verification, simulation and code generation processes.
SDL by nature is a dataflow, hierarchical built, with Extended Finite State Machine (EFSM) as the lowest level defining the behaviour of the system. The system entity is the framework as well as the main structure of the language. The system entity defines the interface with the external world. Next lower level, defined inside system entity, is the block which defines the structure of the specification. Messages are used for communication between different blocks in a system. Messages are also used for communication with the external world. These messages are sent through structures called channels. A block containing one or several processes defined as EFSM. These communicate with each other through signals using signal routes.
Each process possesses an infinite queue in which all incoming signals are stored and consumed according to FIFO (First-In-First-Out), with some exceptions.
If a signal is expected in the current state of the process, the transition related to this event is executed. Otherwise the signal is discarded unless it is saved in the queue by means of the save construct. SDL follows a totally asynchronous model of computation that combines characteristics of processes networks with FSMs. Each SDL process executes concurrently and the specification behaviour of a SDL system is the sum of the individual behaviours of each specification process. SDL
also allows new abstract data types definitions and dynamic creation of processes.
SDL is a modeling language which helps designers express and verify their design ideas in an adequate way. This means that the language is expressive and unambiguous; it has platform-independent semantics, operational semantics and adequate support for modularization.

2.5 SDL limitations

2.5.1 Introduction

SDL is arguably the most successful formal technique used today with widespread usage throughout the software and the telecommunications industries. Part of the reasons for its general adoption is its intuitive graphical notation and excellent tool support. The tool support typically offers capabilities to analyze, design, implement and subsequently test systems, often using combinations of interrelated notations together with SDL such as Message Sequence Charts.
One of the main perceived benefits of SDL over other notations such as UML is the ability to model and reason about, e.g. via model checking tools, detailed behavioural specifications, including real-time behaviours.
SDL has some language aspects for expressing features of timed systems, these are unfortunately inadequate for hard real-time systems development, because they are indeterminable. Since timer expiry results in an input signal being placed in the (possibly non-empty) input queue of the associated agent, these signals can be in the queue any arbitrary time before they are consumed. When a timer is set to t seconds, the interpretation of this timer is in fact an arbitrary time duration dt (dt € [t,∞])). Such a weak interpretation of timers cannot provide enough expressive power to describe the timing behaviour of hard real time systems.
The time mechanism in SDL is heavily affected by the platform-dependent physical clock. Such a platform dependent timing mechanism cannot provide facilities to debug and analyse timing behaviour of a model, because any debugging and analysis observation may introduce extra time passing, which changes the real-time behaviour of the model and leads to unreliable debugging and analysis results.
As a natural consequence of language limitations, associated tools suffer from a lack of precision for dealing with the temporal aspects of specifications and are often unable to enforce or establish the existence of temporal properties.
Typical examples of the properties that a real-time specification language and associated real-time tool support should be able to check for include:

- *deadlock properties* where the real-time specification reaches a state where no more transitions are possible and time progresses indefinitely;
- *livelock properties* where the specification is unable to ever receive messages (signals) from the environment due to continuous internal interactions;
- *invariant properties* that must hold for all executions of the model including real-time invariant properties;
- *non-zenoness of runs* where time in the system does not progress beyond a certain value due to continued (non-time dependent) interactions;

As well as these classical real-time properties, SDL lacks more general properties as e.g. to describe *non-linear properties* such as signal *X* should be followed by signal *Y* within a maximum of *Z* time units.
To achieve this, a precise notion of time in SDL and language features that allow for various timing aspects to be both modeled and subsequently validated by associated tools is required.

3. Introduction to real time operating systems

3.1 General

In general a real time operating system is designed to do the work that almost all real time systems do. The intension of the real time operating system is to replace the control loops, jump, global variables and calls that otherwise would control the execution of an application. Maybe the major advantage with operating system is the substantial decrease in manpower for development of a system. One reason for this is that a large and potentially difficult part of the program has already been written. This even before the project has started. Another reason is the fact that the program can be divided into smaller portions of independent programs, making it easier to write, debug and modify than a big complex program. Real time systems are characterized by the severe consequence if a logic as well as timing correctness property of the system is not met. The real time system is evaluated not only with regards to result produced, but also on the time delay at which the results are presented.

Real time system can be distinguished with respect to following:

- Hard real-time system. The tightness for time deadlines is strict. Exceeding a deadline (too early or too late) may result in a disaster, e.g. pacemaker or the controlling logic of some vital systems in an airplane. The whole system has to cope with all temporal requirements in order to be correct.
- Soft real-time systems. Acting after the deadline, if one exists, may not cause a disaster, but surely influence on the QoS for the output of the system. A telephone system with an audible delay is unpleasant but not critical. Typically in a soft real-time system the requirements are defined as minimal quality of service.

Further a real time system can be spilt into following categories:

- Critical system. This is measured by the consequences of a failure in the system. A major failure causes in the worst case loss of human life.
- Non-Critical systems. Missing a deadline won't harm much, but the system might be totally useless.

The usual approach is to have a combination of the hard and soft real time requirements in the same system, being critical or non-critical. There is no distinct composition for the combination of these four, so any combination in a bigger system is possible. The normal approach is then to define parts of the system as e.g. hard real time with critical outcome if the requirements are not met. Real time

operating systems are used in many different applications, raging from huge PC systems down to the tiniest embedded application. Most real time systems are embedded, e.g. cellular phone, laundry machine.

3.2 Real-time concepts

3.2.1 Triggering the system

There is two common ways of triggering a real time system. If the system is based on time periodic triggering, it's classified as a time triggered system. Instant interaction with the outside environment can't be accomplished, but interaction can occur at certain point in time if the system polls the external devices. One way to implement this would be in a control loop.
The other method is event triggering, and the execution is guided of the non-deterministic occurrences of external and internal events. This results in a timely acceptable instant interaction with the environment. Such a system can be implemented with e.g. ISR's.

3.2.2 Foreground background systems.

When designing a small system, it's possible to design it as a foreground background system, also called control loop program. The whole program executes in one infinite loop, calling subroutines (modules) to get desired action done. This is referred to as the background system. To be able to react on asynchronous events, such as input from the environment, an ISR can execute immediate action. This is referred to as the foreground system. Letting the ISR handle time critical events, assures that the operations are dealt with in a timely manner. If the background system is handling information passed from the ISR, it's not guaranteed that it will be dealt with immediately. It depends entirely on when the background routine get executed. A lot of the high volume consumer products act as a background foreground system, e.g. laundry machines, telephones.

3.2.3 Process synchronization

A process, also called a thread or a task, is a simple program that thinks it has the CPU all to itself. Referring to a task or process in a real time operating system, will have the same meaning. If one have several tasks running concurrently it's called multitasking. The processes run in an infinite loop and execute its program code step by step. Since the system is divided into several processes, some synchronization between them is necessary. The synchronization can be done in several ways. If the synchronization involves several events it can be done with two different methods. Synchronization can occur if one of the events have occurred, this is called disjunctive synchronization. The other one is conjunctive, now all events must have happened in order to be synchronized and further execution can start.

3.2.3.1 Mutual exclusion

Mutual exclusion (often abbreviated to *mutex*) algorithms are used in concurrent programming to avoid the concurrent use of un-shareable resources by pieces of computer code called critical sections.
In the light of process synchronization one can use this method for accessing shared data structures. When e.g. an ISR is writing to a data structure, it has to lock the reader of the data out until all structure are updated. Mutual exclusion can be obtained with:

- Disabling interrupts
- Performing test- and set- operations
- Disabling scheduling
- Using semaphores

The problem is acute because without special care, an interrupt can occur between any two instructions of the non-interrupt code, including the very code used to communicate with the interrupt code. If the critical section is not protected, this can cause severe failures.

3.2.3.2 Disabling interrupts

The easiest way of gaining mutual exclusion is to disable interrupts. This has to be done with care to prevent a long period of time with interrupt disabled. Environment stimuli could then be lost.

3.2.3.3 Semaphore

A semaphore is a protected variable (or abstract data type) and constitutes the classic method for restricting access to shared resources (e.g. storage) in a multi processing environment. They were invented by Edsger Dijkstra mid 1960, and first used in the T.H.E. operating system. [TheFreeEncyclopedia.com]

A semaphore can be a hardware or software flag. In a multitasking environment the semaphore is most likely a variable with a value indicating the status of access to a common resource. Two types exist:

- Binary semaphore.
- Counting semaphore.

The binary can only have 2 values: 0 or 1. The counting can have values in the range of the variable type assigned to it. The kernel needs to keep track of the value of the semaphore and which processes is waiting for the semaphore. Processes who want to wait for a semaphore are set in a wait state and put in the queue for waiting processes. A process desiring a binary semaphore performs a state check. If the semaphore is available it changes the state of the semaphore and continues execution. If the semaphore is occupied the kernel puts the process in a queue waiting for it. The wait state can have a timeout, depending on the support of the kernel. Once the process occupying the semaphore releases it, the kernel checks which process to give access next time. The selection can be based on several options such as:

- Highest priority process
- FIFO queue system

Some resources can manage several accesses simultaneously. Imaging a pool of buffers available to the processes, in such a scenario a counting semaphore, with a maximum value same as the number of buffers, is an option. One “key” for each puffer is available. A task requiring a buffer, checks the semaphore and increments it if a buffer is available. When the process releases the buffer the semaphore is decremented, making it available for other tasks.

3.2.3.4 Messaging

Messages can be exchanged between processes (or ISR) through a kernel services. In most cases the message is only a pointer to an allocated area in the memory containing the data to be sent, this is called a mail box or a message exchange. The kernel provides services for posting the message and for receiving the message. A list for waiting processes is associated with the mailbox, in the case that several processes want to receive a message thorough the mailbox. Tasks polling for a message in an empty mailbox are suspended and placed in the waiting list. Also in this service a timeout for the waiting period should be able so specify. Messages can also be completely copied (including all data) and sent to the other process, also done through the kernel service. If a process should be capable of receiving several messages almost simultaneously, a message queue is used. A message queue is a list of messages waiting to be consumed by the receiving process. A normal method of organizing the queue is first in first out (FIFO), meaning that first message posted is the first to be read of the receiver.

3.2.4 Resource

A resource is a entity used by a process, it can be a variable within the system or a I/O device such as printers, disks. Resources can be shared amongst several processes or it can be dedicated to a single process.

3.2.4.1 Shared resources

A resource accessed by several processes is a shared resource. A shared resource should normally be given a special attention when accessing it. Normally a semaphore must be used to control the access to the resources, gaining control of the use.

3.2.4.2 Reentrant functions

A reentrant function is a function that can be used by more than one task without fear of data corruption. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables (i.e., CPU registers or variables on the stack) or protect data when global variables are used. An example of a reentrant function is shown in listing below.

```
void strcpy(char *dest, char *src){
        while (*dest++ = *src++) {
                ;
        }
        *dest = NULL;
}
```

Because copies of the arguments to *strcpy()* are placed on the task's stack, *strcpy()* can be invoked by multiple tasks without fear that the tasks will corrupt each other's pointers.
An example of a non-reentrant function is shown below. *swap()* is a simple function that swaps the contents of its two arguments. For sake of discussion, I assumed that you are using a preemptive kernel, that interrupts are enabled and that *Temp* is declared as a global integer:

```
int Temp;

void swap(int *x, int *y) {
        Temp = *x;
        *x = *y;
        *y = Temp;
}
```

We can make *swap()* reentrant by using one of the following techniques:

- Declare *Temp* local to *swap()*.
- Disable interrupts before the operation and enable them after.
- Use a semaphore.

If the interrupt occurs either before or after *swap()*, the x and y values for both tasks will be correct.

3.2.5 Multitasking

A technique used in an operating system for sharing a single processor between several independent jobs [TheFreeDictionary.com].

Since task is the common name for an independent portion of an application, when referring to real time operating systems, we'll use the word task instead of process in this chapter.
The idea of multitasking is to share the central processing unit (CPU) between different tasks in a system. The different tasks can be seen on as multiple background systems. This approach ensures that the system get maximized throughput since the CPU always get to run a ready task. Another benefit of multitasking, especially in complex systems, is the ability to split the jobs to be done in smaller portions called tasks. This makes it easier to manage a system because each module is an independent program. Maybe the biggest benefit of multitasking is the programmer's ability to manage complexity inherent in real-time applications

3.2.5.1 Tasks state

During the execution of an application, the tasks change state. The current activity of the task partly defines the state of the task. A task can be in one of the following states.

- New, a new task is dynamically created.
- Running, the task is executing in the CPU.
- Waiting, the task is waiting for an event to occur, bringing it out of this state.
- Ready, the task is waiting for CPU time.
- Terminated, the process has finished execution for the lifetime of the application.

The task control block (TCB) holds information about the current state of all tasks. If referring to the ready list simply means the collection of all tasks that have the ready status in the TCB.
The following figure visualizes the transition between the different states, and the possible action to trigger the transition from one state to another state.

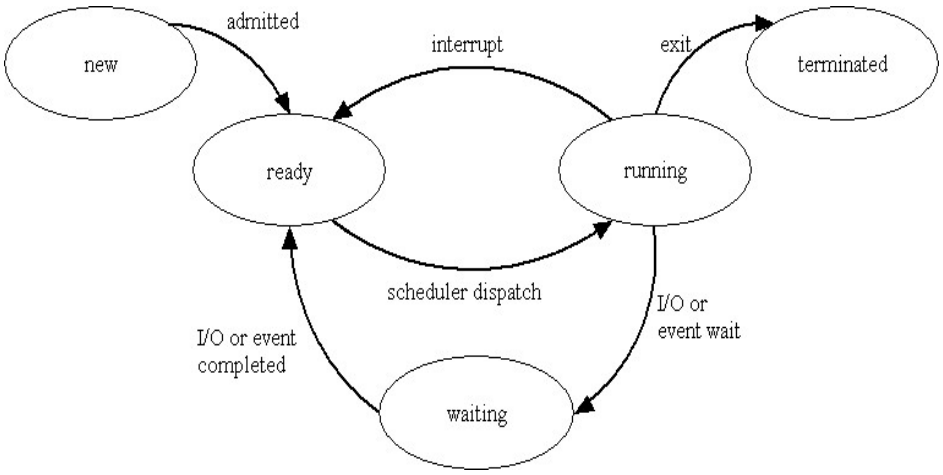

Figure 1

A representation of all different states a process can be in.

3.2.5.2 Context Switch

When a multitasking kernel decides to run a different task, it simply saves the current task's *context* (CPU registers) in the current task's context storage area – its stack. Once this operation is performed, the new task's context is restored from its storage area and then resumes execution of the new task's code. This process is called a *context switch* or a *task switch*. Context switching adds overhead to the application. The more registers a CPU has, the higher the overhead. The time required to perform a context switch is determined by how many registers have to be saved and restored by the CPU.
The following scenarios may generate a context switch:

- A task switching from the running state to waiting state (e.g. waiting for a signal).
- A task switching from the running state to the ready state (e.g. occurrence of an interrupt).
- Tasks switching from the waiting state to the ready state (e.g. reception of a signal).
- Task terminates.

For scenario one and four a context has no choice of scheduling. A new task must be selected to execute. For scenario two and three there are some possible outcomes for the scheduling. This is what differs preemptive and non-preemptive scheduling. The topic is described later in the text.

3.2.5.3 Micro kernel

An operating system kernel is the portion of the operating system that is common among all other operating system services and applications. Kernels can be specialized according to the applications that they support. One such specialization is that for the real-time system. A real-time system, given its strict temporal requirements, requires a specific set of services that a kernel must provide. [CS423, Advanced Operating Systems Research in Real-Time System Kernels Tanya L. Crenshaw]

The kernel is the heart of the operating system, being real time or not. Its job is to manage the CPU time between the tasks, as well as the communication between them. The fundamental service provided by the kernel is context switching. A kernel will certainly add overhead to your system, but normally the benefits are much more distinctive. Since each task has its own stack space the RAM will be eaten up quite quickly. The kernel also consume CPU time to get the job done, normally this is about 2 and 5%. In a small chip controller it is of great importance to keep the kernel as small as possible, this will usually be in the cost of the supported functionality.

3.2.5.4 Non-Preemptive micro kernel

Non-preemptive or cooperative multitasking, also called, requires the tasks to explicitly give up the CPU. The asynchronous events can still be handled by the ISR, but the ISR always return back to the last executing task. If the ISR makes another task, with higher priority than current, ready to run, it still has to wait until the current process gives up the CPU. To get the illusion of concurrency the release of CPU must be done frequently from the process. Some of the advantages with non-preemptive kernel include:

- Low interrupt latency.
- Non-reentrant functions can be used. The process owns The CPU and can finish executing the non-reentrant function.
- Less need of semaphore for access control to recourses. Since the process itself controls when to release the CPU it's easier to control sheared recourses, this is not the case all time.

The greatest disadvantage with non-preemptive kernels is the responsiveness. If e.g. an ISR makes a higher priority task ready to run, it still has to wait until the current task gives up the CPU. To summarize, a non-preemptive kernel allows each task to run until it voluntarily gives up control of the CPU. An interrupt will preempt a task. Upon completion of the ISR, the ISR will return to the interrupted task. Task-level response is much better than with a foreground/background system but is still non-deterministic. Very few commercial kernels are non-preemptive.

3.2.5.5 Preemptive micro kernel

In preemptive systems, the kernel scheduler is called with a defined period, each *tick*. Each time it is called it checks if there is a ready-to-run task which has a higher priority than the executing task. If that is the case, the scheduler performs a context switch. This means that a task can be preempted - forced to go from executing to ready state - at any point in the code, something that puts special demands on communication between tasks and handling common resources. Using a preemptive kernel solves the problem where a high priority task has to wait for a lower priority task to yield the processor. Instead, when the high priority task becomes ready to run, the lower priority task will become preempted, and the high priority task can start to execute. A preemptive kernel is used when system responsiveness is important, most commercial real-time kernels are preemptive.

3.2.6 Deadlock

A deadlock can be a disaster for a critical system, but have only might only have irritating effects on a non critical system. A deadlock occurs if two processes are waiting for the other task to release e.g. a semaphore. Say two semaphores are needed to perform an operation. Task 1 gets semaphore A and tries to get semaphore B. Task 2 takes semaphore B before task A. Now task A waits for task B to release semaphore B, and task B wait for task B to release semaphore B. Resulting in a deadlock. Deadlock can be avoided in several ways:

- Acquire all resources before proceeding.
- Acquire the resources in the same order.
- Release the resources in the reversed order.

The consequence of a deadlock can be minimized if a timeout is specified when waiting for a semaphore. This approach release the system from the deadlock occurred, even though the consequence could be fatal.

3.2.7 Scheduler

If the CPU becomes idle, the RTOS must select one of the other tasks in the ready queue for execution. If none is ready the idle task (always ready) is switched in. The selection of next task to execute can be based on different algorithms.

3.2.8 Task priority

A priority is assigned to each task. The more important the task, the higher the priority is given to it.

3.2.8.1 Static priorities

Task priorities are said to be static when the priority of each task does not change during the application's execution.
Each task is thus given a fixed priority at compile time. All the tasks and their timing constraints are known at compile time in a system where priorities are static.

3.2.8.2 Dynamic priorities

Task priorities are said to be dynamic if the priority of tasks can be changed during the application's execution; each task can change its priority at run-time. The priority of a task is a function of time. The longer in ready queue, the higher is the priority given to it. Another advantage is that if a high priority task trying to access a resource hold by a low priority task, the low priority task can get the same priority as the other task.

3.2.8.3 Assigning task priorities

Assigning task priorities is not a trivial undertaking because of the complex nature of real-time systems. In most systems, not all tasks are considered critical. Non-critical tasks should obviously be given low priorities. Most real-time systems have a combination of soft and hard requirements. In a soft real-time system, tasks are performed by the system as quickly as possible, but they don't have to finish by specific times. In hard real-time systems, tasks have to be performed not only correctly but on time.
An interesting technique called Rate Monotonic Scheduling (RMS) has been established to assign task priorities based on how often tasks executes. Simply put, tasks with the highest rate of execution are given the highest priority.
RMS makes a number of assumptions:

1. All tasks are periodic (they occur at regular intervals).
2. Tasks do not synchronize with one another, share resources, or exchange data.
3. The CPU must always execute the highest priority task that is ready to run. In other words, preemptive scheduling must be used.

Given a set of *n* tasks that are assigned RMS priorities, the basic RMS theorem states that all task HARD real-time deadlines will always be met if the following inequality is verified:

$$\sum_i \frac{E_i}{T_i} \leq n * \left(2^{\frac{1}{n}} - 1 \right)$$

Figure 2

where, *Ei* corresponds to the maximum execution time of task *i* and *Ti* corresponds to the execution period of task *i*. In other words, *Ei/Ti* corresponds to the fraction of CPU time required to execute task *i*. Table 1 shows the value for size $n(2^{1/n}-1)$ based on the number of tasks. The upper bound for an infinite number of tasks is given by *ln(2)* or *0.693*.

This means that to meet all hard real-time deadlines based on RMS, CPU utilization of all time-critical tasks should be less than 70 percent! Note that you can still have non-time-critical tasks in a system and thus use 100 percent of the CPU's time. Using 100 percent of your CPU's time is not a desirable goal because it does not allow for code changes and added features. As a rule of thumb, you should always design a system to use less than 60 to 70 percent of your CPU.

Number of Tasks	**$N(2^{1/n}-1)$**
1	1,000
2	0,828
3	0.779
4	0,756
5	0,743
.	.
.	.
.	.
Infinity	0,693

Table 2

RMS says that the highest-rate task has the highest priority. In some cases, the highest-rate task may not be the most important task. Your application will thus dictate how you need to assign priorities. RMS is, however, an interesting starting point.

3.2.9 Scheduling algorithms

3.2.9.1 First come, first served (FCFS)

Surely the simplest scheduling algorithm discussed in this document. Tasks ready to execute are kept in a FIFO queue. When a process gives up the CPU, either by requesting some resources or explicitly gives up the CPU, the task first in the queue gets the CPU. A task going from wait to ready state enters the queue in the tail. A task switch in from the front of the queue is removed from the queue. FCFS is non-preemptive, as soon as a task gets the CPU it owns it until it terminates or entering a state different from ready. This algorithm is too simple to be used for implementation of SDL. One has no control over time critical operations.

3.2.9.2 Shortest job first (SJF)

Another non-preemptive algorithm is the SJF. The latter's next CPU burst are examined and the one with the shortest next burst are allowed to execute. If two processes have the same burst length, FCFS are used for selection of these two. In this manner the through put of the CPU are increased, because it reduces the waiting time for short tasks more than it increases the waiting time for longer tasks. As a result the average waiting time decreases. The difficulty in this method is to predict the length of the next burst in the short term CPU scheduling as in embedded systems.

3.2.9.3 Priority

A priority is associated with each task. The CPU is allocated to a task in the ready list, having the highest priority. If two or more tasks have equal priority, the choice of task is based on FCFS. Priority scheduling can either be preemptive or non-preemptive. Most embedded systems use preemptive scheduling.
Task priorities can be set in two different ways:

- Static Priorities. The priority of a task is unchanged during the lifetime of the application. A static priority is given at compile time.

- Dynamic Priorities. The priorities of a task can be changed during execution time if desirable. If an external event which happens rarely must be dealt with immediately, the task handling the event gets top priority when the event occurs. This is a desirable feature to have in a real-time operating system for implementation of SDL.

3.2.9.4 Round robin (RR)

RR scheduling is similar to FCFS, but preemption is added. The algorithm is developed for time sharing systems. A time slice is defined, named system tick, normally between 10 to 100 milliseconds. Every time the system tick occurs, the dispatcher switch in a new task. This task is the first task at the head of the ready queue. All tasks in ready state are kept in a FIFO queue. If a task is finished before the system tick occurs the task voluntary gives the CPU to the dispatcher, the dispatcher switch in a new task. If a task still have code to execute at the occurrence of the system tick, the dispatcher switch in a new task and the other task is placed at the end of the ready queue.
Summarized the following situations give a task switch:

- The current task doesn't have any work to do during its time slice or
- The current task completes before the end of its time slice.

The performance of the RR depends heavily on the length of the system tick. The longer lasting the system tick is, the more identical it becomes FCFS. A long system tick gives no control over timing, tasks must wait a long time before they can execute. If the system tick becomes to short, the added overhead for the dispatcher to run becomes apparent, too much of the CPU time if used just for task switches.

3.2.9.5 Multilevel queue

Task with similar characteristics or same priority are placed in a common group. Several different groups can exist within a system. Every group has its own private scheduling algorithm. It isn’t necessary to use the same scheduling algorithm for the groups, different group can use different algorithms. Ones a task is placed in a queue, done at entry of the system, it must stay in the group for the lifetime of the application.
A scheduling algorithm must be chosen between the groups as well. Normally the priority scheduling or time slicing is used.

3.2.9.6 Multilevel feedback

The difference between multilevel queue and multilevel queue feedback is the possibility for the task to dynamically change process group during execution. If a task is using to much CPU time it can be moved down to a group with lower priority, and a task with short CPU bursts can be moved up to a higher priority group. This aging approach prevents starvation of a task. All tasks get the ability to execute.
Following properties make multilevel feedback scheduling:

- The number of queues.
- The scheduling algorithm for each queue.
- The method used to determine when to upgrade a task to a higher priority queue.
- The method used to determine when to demote a task to a lower-priority queue.
- The method used to determine which queue a task will enter when the task needs service.

Multilevel feedback scheduling is the most general scheduling algorithm. The drawback is the cost of time and memory due to the complex decisions that has to be done by the scheduler.

4. Essential RTOS properties to describe SDL systems

4.1 Introduction – Simplified SDL

An SDL system can be described as process instances that communicate by sending signals to each other or to the environment. Each single process can be viewed as an autonomous finite state machine, working concurrently with other processes, co-operating with other processes or the environment through discrete messages (signals). Depending on the next input, the process performs a transition; that may include many actions, and finally moves to a new or same state. Its next state may be determined by decisions in the actions. In SDL, a state is the only location where input from the queue can trigger a transition. All communication in SDL is asynchronous.

4.2 Process states and transitions

In SDL all processes are supposed to run simultaneously, in an embedded platform such behaviour is impossible to achieve. Thus, all processes have to share the same processor, in a pseudo-parallel manner. In that way only one process is executing at any time. All others processes are either ready-to-run or blocked. If an external event or a process timer expires, the kernel scheduler have to check if the event enabled a higher priority process, and if it did this process will be dispatched, and a transition will take place.
This implies that scheduling is needed, preferably a pre-emptive to handle spontaneous events, and with priority.

4.3 Signals and queues

Further, each process should have a unique address (*Pid*). A signal is always assumed to carry the address of both the sending and the receiving process, in addition to possible data values. The receiving process thus always knows the address of the sending process.
A process should have an infinite input queue, where incoming signals are to be queued. When a signal has initiated a transition, it should be removed from the input queue (consumed). A process should also have the possibility to save signals for later use, if no input under the current state contains the received signal.

4.4 *Timing*

All systems implementing the conceptual basis of SDL should contain methods for timer management; because in most system descriptions certain time constraints are commonly used. SDL has the *timer* construct for this. The timer is an object owned by a process, which is able to generate a timer signal and put this signal into the input queue of the process. SDL defines a set of operators for timers, with these we have the ability to set, reset and to check if timers are active (running).

4.5 *Triggers & transition elements*

If an embedded real time operating system has implemented the features described above, and in addition has implemented all the triggers, and transition elements; it's should be capable to use for implementing systems described with SDL. A real time kernel whose purpose is to implement the SDL behaviour, need to implement the SDL semantic, the SDL/PR syntax and SDL data types are of less importance.

Not all triggers are applicable for use in embedded systems, as is true for "continuous signals". This will be described in chapter 7 "omitted SDL properties".

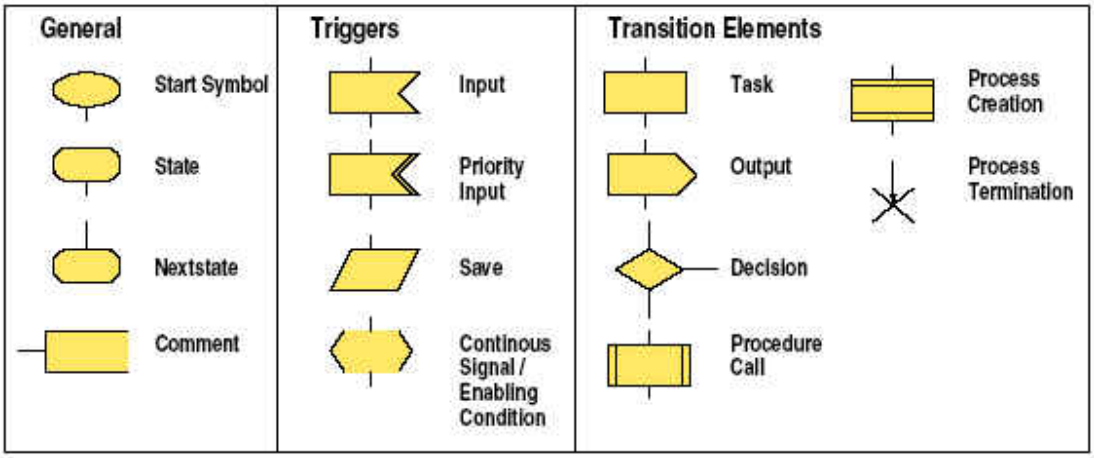

Figure 3 List of SDL symbols.

5. Review and analysis of some existing RTOS

5.1 OSE Epsilon for AVR

OSE is a RTOS kernel developed by ENEA Embedded Technology in Sweden. The kernel is ported to different microcontrollers. OSE for AVR is exclusively designed to fit the peculiarities of the AVR processor family.

5.1.1 Properties of OSE

ENEA Embedded Technology claims that OSE for AVR is a fast, compact real-time operating system for AVR microcontrollers. The size of the kernel is estimated to approx 1kbyte RAM, depending on options set in the configuration files. Direct process-to-process message passing is the hallmark of OSE Systems' RTOS, the same as with SDL.
OSE build processes as independent tasks, just like a subprogram. Each process can be written as if it has the entire CPU on its own. OSE has the ability to run static processes, created at compile time. It also has the feature to create, run and kill dynamic, predefined, processes at runtime. Each process is an independent program which is assigned a private area of memory.
OSE has defined to types of processes, independent of dynamic or static creation. These to types are prioritized and background. The prioritized processes are, as the name indicate, assigned a priority dependent of the importance of the task. The priority is just a number ranging from 0 to 31, where 0 reflects the highest priority. Priority is set by the user of the RTOS. A process can be started and stopped during program execution.
The dispatcher in OSE will assure that the process with highest priority, supposed it's in ready state, are given CPU time to execute.
Another quality of OSE is its "pre-emptive" dispatcher, meaning that the dispatcher has the ability to stop the current process after the next assembler instruction (even within a system call) and move the execution to another process.
A background process is not assigned any priority. All background processes are put in a pool, and given the possibility to execute when all prioritized processes are in a wait state. The dispatcher determine which process are given the opportunity to execute after the principle of "round robin", the user can determine the length of the time slice interval.
The last category of process is interrupt processes. This is hardware activated and can run inside or completely outside the OS. The user can determine which approach to use for the interrupt process. Only one process can be created for each hardware interrupt vector.
A special case of interrupt processes are timer interrupt processes. The user has the ability to set the number of elapsed system ticks between each call to the

process. The number is hold in an 8-bit variable, ranging from 1 to 255. The timer process has the same priority as the interrupt source.
OSE systems do not recommend the use of global variables used by more than one process, even thou global variables can be used. To access the global variables semaphores is used, available through the kernel as a system call. Before entering a global variable the interrupt must be disabled, enabled again when done operating on the variable. The use of global variables are not recommended from the inventors side, instead they stress the use of signals as the safest way to do communication between different processes. A signal is no more than a message passed from one process to another, with or without data included in it. All signals are put in a signal buffer. This buffer can only be accessed by one process at a time. Once the sender process has sent the signal, it has no more privileges to access the signal. The owner of the signal buffer is the process that receives the signal. Only the owner can do operations on the buffer.
The signal buffer pool is administrated by the OS. Eight different user predefined sizes are available. When the user defines the size needed for the signal, the OS releases data memory, from the message pool, according to the closest higher predefined size. The reason for this approach is to prevent memory leakage and fragmentation. The signal memory area is released from the pool and handed over to the process, the process sends the signal to another process. The receiving process consumes the signal and releases the memory area back to the signal memory pool.
The receiving process can scan thru the signal queue looking for one or several signals at a time. The first signal in the queue matching the parameters is consumed.

5.1.2 Shortcomings of OSE

At a glance OSE seems well suited for describing SDL systems in ANSI C code. Especially the property of sending messages between processes makes it well suited for SDL implementation. There are after all a few disadvantages found in the RTOS. One of the main drawbacks is the size of the kernel and the amount of memory claimed by processes and signal buffers. The total memory needed is close to the total memory available on the biggest 8-bit AVR controllers. The kernel requires approximately 1 kbyte RAM and each process need 96 byte, plus stack and pool size. Minimum code size of OSE for AVR is approximately 6 kbyte ROM. The RTOS is designed for a microcontroller set up for a capability of addressing up to 16 Mbyte data memory and 8 Mbyte program memory. This amount of memory is far beyond the size of our embedded target.

5.2 PR_RTX

PR_RTX is produced by Progressive Resources LLC exclusively for the CVAVR compiler.
They claim that the kernel is a lean, mean task switching machine for the some of the AVR processors.

5.2.1 Properties of PR_RTX

The RTOS is initialized and implemented by the inclusion of a code file and a control block in the heading of the program file. All processes should be written as independent tasks.
All control structure (TCB's) is created and held in the program memory. The RTOS are not supporting dynamic allocation or de-allocation. This means that deleted processes still take up RAM in the memory, and it makes the creation of dynamic processes impossible. PR_RTX support the feature of interrupts, making none or small impact on the timing as long as the interrupt process execution time is shorter than one system tick since there is only one level of interrupt. The user has the ability to deal with an interrupt in the standardized ISR, making the RTOS vulnerable for heavy user code loaded ISRs. If so is the case it will certainly make an impact on the interrupt latency.
The RTOS can be run in two modes, Round Robin or task priority based.
Timer 0, independent of processor, is used as the default base for generating system ticks. As an alternative tick base, any interrupt routine can be used to generate system ticks. The user will then have the responsibility to ensure that a system tick happens on a regularly time interval. When Timer 0 creates an interrupt (system tick), or a task enters a blocked state, the kernel can make a task switch. Either it switches in a ready task or the idle task is selected. Main function is defined as the idle task. When a context switch is due the kernel stores away the stack pointers for the task to shift out, and load the stack pointers for the new task.
When running in RR mode, it is important that the tasks give up the processor as often as possible, with either PR_End_Task() or one of the PR_Wait functions.
In priority mode the task with highest priority are given the chance to run ones pr tick. Task with a higher priority are not able to pre-empt task with lower priority. Progressive Resources propose the use of shorter tick rate, alternatively to handle high priority task on an interrupt basis, as a substitute for the drawback on not having pre-emptiness.
During the creation of a task, the user has the choice of setting the task to active or dormant.

The following system calls are implemented and are of interest in the scenario of implementing a SDL system:

- PR_Wait_Ticks(ticks), ticks denotes number of system ticks the process is put to sleep. Ticks is a char type, so 255 is the maximum number of wait ticks.
- PR_Wait_Semaphore(ticks), ticks is the maximum time to be put in a sleep state, waiting for a semaphore. If the zero is used, the timeout function is disabled.
- PR_End_Task(PR_active or PR_inactive), returns the control to the task switcher. If PR_active is used, the task is in a ready position and will be executed as soon as it's time. If PR_inactive is used, only another task can put it in a ready state again
- PR_Stop_Task(Task_number), puts Task_number in a dormant state. The task can be re-enabled, starting from the end point of execution or totally re-started again. This lies under the control of the system call PR_Enable_Task().
- PR_Enable_Task(Task_number, PR_continue or PR_restart), are used to enable other tasks out of a dormant stat. Using PR_continue as a input parameter, forces the task to start from last executed statement. PR_restart forces the process to do a complete restart.
- PR_Query_Task(PR_Task_Number), gets the state of another task. Possible replies are dormant, ready, waiting semaphore or waiting timer.
- PR_Send _Semaphore(Task_number, Semaphore), sends a semaphore to another task. If the task is waiting for a semaphore, it's put in a ready state. If it's not waiting for a semaphore this call have none effect. The semaphore may contain a value between 1 and 15.
- PR_Query_Semaphore(), checks the source for it's awakening. Possible outcome is awakened by timer timeout or acceptance of a semaphore.

5.2.2 Shortcomings of PR_RTX

First of all PR_RTX has very limited features. The non-existent possibility to allocate memory at runtime is a substantial shortcoming. The ability to create dynamic processes at run time is of vital necessity for an correct implementation of SDL.
Further more the RTOS uses semaphores as a communication base between processes. This doesn't comply with the strict semantic of SDL, where signals passed form process to process is the only inter process communication available. A signal in SDL can theoretically contain an unlimited amount of information, the send semaphore system call in PR_RTX can only contain a value from 0 to 15. Resulting in a divergence in the amount may needed and actually offered in passing a signal containing information. With the already mentioned shortcoming it seems quite obvious that PR_RTX is not suited for implementation of a SDL system. Further discussion seems unnecessary when fundamental properties are not supported in the RTOS.

5.3 C Micro

C Micro is developed by Telelogic Tau as a RTOS for their code generating tools. It's used to realize SDL system in the ANSI C language. The RTOS is a library with a configurable SDL kernel. Telelogic Tau is one of the leading companies in tools for system description and code generating.

5.3.1 Properties of C Micro

The Cmicro Library consists of a configurable SDL kernel together with all the necessary SDL data handling functions. The collection of C functions and C modules make up the so called SDL machine. The Cmicro kernel can be scaled down to 4kByte. The RTOS comes in a pre-emptive kernel or a non pre-emptive kernel version. The run time model in Cmicro is such that there are global variables used in the generated C code and the Cmicro, library variable and function names have to be unique within the whole system. In principle no (automatic) dynamic memory allocation is possible. However this is a truth with modifications. For signal instances a static buffer is allocated at compile time. The user has to specify the size of this buffer. If a process wants to send a signal, it requests a memory area from the buffer. The memory area is released, and when the signal is consumed the memory area is handed back to the buffer. The predefined type TIMER exists, and acts just like a signal in the system. If a process requests a memory area from the buffer when the buffer is empty, dynamic memory allocation can be used. This option has to be declared before compilation. Dynamic allocation can't be used together with the preemptive kernel. Before compilation the maximum number of instances for a process must be specified. This is due to the fact that no dynamic allocation is possible. All processes are crated at compile time, at the number of instances specified by the programmer. Processes that are not meant to start at the initial start of the application are put in a dormant state. When a process is to be "dynamically created" it only changes state from dormant to ready. This behaviour simulates dynamic process creation. If a process is supposed to be terminated, it won't release the memory area, it only enters the dormant state. No formal parameters can be past at process creation.

5.3.2 Shortcomings of C Micro

C Micro, even design for SDL implementation, doesn't support the feature of dynamic memory allocation in combination with pre-emptive kernel. This is necessarily not a shortcoming, as long as you specify the maximum number of a process instances before compilation. The compiler allocates memory area for the total number of processes, but not all are instantiated. When a "dynamic" process is created it gets an already allocated area of memory. This approach is time effective since dynamic allocation can be avoided. The drawback with this approach is the immediate need for memory. Using a real dynamic allocation is less memory consuming, an important property when developing for an embedded target, even though more time is consumed during allocation. Memory is a scarce resource in our target. Another feature needed to implement the behaviour of SDL is the possibility to pass parameters to a process at time of creation. Cmicro doesn't support this feature due to the fact that all processes are created at compile time.
Maybe the main reason for not using Cmicro is the cost per license. It is far beyond the cost that HiA is willing to spend.

6. Design consideration for the SDL REFLEX RTOS kernel

6.1 Introduction – simple and unambiguous

The primary motivation, when designing this real time operating system kernel was to simplify the development process of real time systems for embedded systems. To achieve this goal the most reasonable way was to adopt the behaviour of a simple and unambiguous formal description language. A thorough study showed that SDL was the most appropriate at present time; not only because of its simple conceptual basis, but also because toolmakers like Telelogic, and PragmaDev provides drawing and analyzing tools for this language. Thus, complete system could be drawn and analyzed in detail, for subsequently to be implemented in the AVR target processor on top of the micro kernel running on top of the micro kernel real time operating system.

6.2 Optimization

Animportant issue regarding the development of the micro kernel real time operating system is to decide what performance property is the most critical. Two disjunctive approaches are of current interest, either to code optimize or to speed optimize. These approaches can partly be determined with a compiler setting, and partly forced through the structure of the program.
For example by declaring inline functions, one can direct the compiler to integrate that’s function code into the code of its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; usually the code becomes larger with function inlining, depending on the particular case. Inlining of functions is an optimization and it really "works" only in optimizing compilation.

6.3 Critical stack

In embedded platforms stack overflow is very critical. If overflow occurs the system will easily crash, however to ensure that a system don't run out of stack is almost impossible. To reduce the chance of stack overflow, it's common to observe the stack size during testing, and afterwards to allocate the largest stack size ever observed by some safety margin. A larger safety margin would provide better insurance against stack overflow, but for embedded processors used in commercial products such as sensor network nodes and consumer electronics, the degree of over provisioning must be kept small in order to minimize per-unit product cost. Thus, the decision should be closely considered. Testing-based approaches to software validation are inherently unreliable, and testing embedded software for maximum stack depth is particularly unreliable because its behaviour is timing dependent: the worst observed stack depth depends on what code is executing when an interrupt is triggered and on whether further interrupts trigger before the first returns.

6.4 Decision of programming language

6.4.1 Decision factors

The decision of what language to use to implement the operating system is a difficult one. Many factors must be considered and different weights given to each of them. The factors relevant to a language decision probably include at least:

- Efficiency of compiled code
- Source code portability
- Program maintainability
- Typical bug rates (per thousand lines of code)
- The amount of time it will take to develop the solution
- The availability and cost of the compilers and other development tools
- Your personal experience with specific languages or tools

6.4.2 High-level language advantages

The advantages of using a higher-level language, for implementing operating systems are the same as those accrued when the language is used for application programming:

- The code can be written faster,
- is more compact,
- is easier to understand,
- is easier to debug.
- Less lines of code = less chance for errors
- More Portable

In addition, improvements in compiler technology will optimize the generated code for the entire operating system by simply recompile. Finally, an operating system is far easier to port to some other hardware if it is written in a high-level language.

6.4.3 High-level language disadvantages

The major claimed disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. Although a modern compiler can perform very complex analysis and apply sophisticated optimizations that produce very good code. In addition, only a small amount of code is critical to high-performance; the memory manager and the CPU scheduler.

6.5 Kernel Implementation

In the decision process, the above problems have been closely weight, for and against. Since the target for the operating system primary is the AVR 8-bit RISC microcontroller family, we've decided to focus on program size, and memory usage. Thus, optimization done will be to reduce code size, and not for speed, though both would be preferable. If more program memory were available, time optimization would be preferred.

In order to preserve stack safety we've made some effort. As describe above the worst observed stack depth depends on what code is executing when an interrupt is triggered and on whether further interrupts trigger before the first returns. In order to overcome this problem we've decided not to allow interrupt nesting, this involves interrupt latency in the system, which occasionally might to be a problem but solves unpredictable stack usage. In order to do this, all interrupts have to be handled by the kernel. From SDL's point of view, there isn't such a thing as interrupts, neither is use of global variables. Thus, in order to deal with these features we introduce interrupt-signals. Interrupt signals are signals which are sent from the environment to a process listening for these signals.
When an interrupt occurs, the system will immediately dispatch a signal to whatever process listening for this signal, and at the same time give the listening process the highest priority and set ready to run. In such a way that the listening process will execute straight after the interrupt handler terminates, to avoid further latency.
Another important issue regarding stack usage is to inform the users of the potential problem regarding stack; and aware them of using recursion in their systems.

The micro kernel does neither use inline functions or templates, in order to keep the usage of program memory as small as possible.
The SDL REFLEX micro kernel has implemented the usage of dynamic task creation, so the users should roughly estimate how many dynamic tasks which possible could run concurrently, in order to estimate if it's possible to run the systems on the chosen controller, or if he has to add extended memory.
Since the resources in the AVR family microcontrollers are limited, the kernel can't implement all of SDL's functionality, therefore some features have to be omitted, what features which will be omitted is described in the next chapter.

7. Functional description of the SDL REFLEX microkernel

7.1 Introduction

SDL REFLEX is the micro kernel of a real time operating system. The micro kernel provides the most essential functions to write programs for embedded computer systems.
The kernel is especially designed to implement systems described in SDL – "The Specification and Description Language" recommended by ITU-T. SDL REFLEX is primary implemented for Atmel's AVR 8-bit RISC microcontroller family, and is written to compile with the GNU ANSI C compiler for AVR v.3.3.

7.2 Overview

SDL REFLEX is a small, portable and efficient pre-emptive microkernel real time operating system. It has been designed specifically for resource-constraint embedded systems. SDL REFLEX controls access to system resources and schedules program processes according to process priority. By introducing the process concept, the internal system operation is coordinated and synchronization can be performed between processes. SDL REFLEX is a pre-emptive operating system, which means that the system can interrupt any process at any point in the execution, and let another process continue execution.
The processes communicate with each other through signals. A signal is actually a message which is sent from one process to another in order to inform the receiver of an event, or to send some data. Each process has its own FIFO queue for incoming signals, in which received signals are stored. When a signal is consumed, it's removed from the FIFO queue. All the actions a process performs are usually responses to received signals. All interaction with the SDL REFLEX microkernel is through a set of system services.

7.3 Most prominent kernel features

An SDL system is defined by the behaviour of the processes it consist of and how they are interconnected. By, ignoring many details we can describe an SDL system at run-time as process instances that communicate by sending signals to each other or to the environment. Each single process can be viewed as a finite state machine, acting on input. Depending on the next input, the process performs a transition; that may include many actions, and finally moves to a new or same state. Its next state may be determined by decisions in the actions. In SDL, a state is the only location where input from the queue can trigger a transition. All communication in SDL is asynchronous.

Down below is a list of the system services that has been designed to “reflex” the behaviour of SDL. These are only the most prominent features.

7.3.1 CREATE

SDL:

In SDL the create statement is associated with the creation of dynamic process instances. In SDL parameters may be passed to the created process instance.

SDL REFLEX:

This function tries to create an instance of a given process-type; this might fail if no more memory is available, and if this happens a NULL pointer is returned. If the creation of the process was successful, a pointer containing the handler for this function is returned, in SDL REFLEX this is referred as the process Pid. Unlike the SDL definition, in which CREATE only is associated with the creation of dynamic processes, the SDL REFLEX create has be used to create both static and dynamic processes. Further, in SDL REFLEX the kernel needs to know how much workspace the new process is assumed to use and the priority of the process, this information is to be passed as parameters to the create function. In SDL REFLEX as in SDL, it is possible to pass creation parameters, if this is to be done, CREATE_WITH_PARAMS function should be used.

```
Pid CREATE (                    void (*proc_code)(void *),
                                u08 priority,
                                u16 stackSize)

Pid CREATE_WITH_PARAMS (        void (*proc_code)(void *),
                                u08 priority,
                                u16 stackSize,
                                u08 paramSize,
                                void*param);
```

Figure 4 Create

7.3.2 INPUT

SDL:

In SDL all signals reaching a process are stored into a FIFO (first in first out) queue associated to every process instance.
When performing an input:

- The first signal in the signal queue is removed from the queue.
- The values of the signal parameters if any; are assigned to the variables specified in the input symbol.
- The Pid of the process which transmitted this variable is stored into the predefined variable SENDER.
- The receiver process performs the transition.

SDL REFLEX:

This function does not really perform the assumed action described in SDL, but when used in association to WAIT_SIGNAL they comply with the SDL input concept. The SDL REFLEX function INPUT, is used to select a set of signals, which the process is going to wait for. At the end of the signal list, a special terminator should be applied, to inform the kernel of where to stop scanning for arguments. In SDL states, all processes have an infinity sized signal-queue. The select list is also associated with the process, each time the process is to be resumed, the process scans through its input queue to see if any of the signals in the queue exist in the select list. If it does, this signal is to be consumed. And the process is to perform a transition.

- If *priority signals* is present in the list of *input signals*, these will if present be picked from the input queue before other signals, even though they don't appear to first in the FIFO queue.

void INPUT (SIGNAL signal,... /* sig1,sig2,…,sigN, END_LIST */);

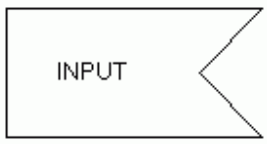

Figure 5 Input

7.3.3 SIGNAL

SDL:

In SDL a signal is transient asynchronous event, transmitted by a process instance to another process instance. A signal may carry one or more parameters.

SDL REFLEX:

In SDL REFLEX a *signal* has to be created prior to be used. And to create a new signal, simply call the system service NEW_SIGNAL with subsequent parameters.
This function is executed whenever a new signal is needed, it's not certain it will succeed. If it proves a success, memory is allocated. To be able to estimate how much memory which is to be used with the signal, the function needs to be informed of the size of all the parameters which is to be sent with the signal.
If this call does not succeed in allocating memory, a NULL pointer is returned, instead of real signal pointer.
In addition to the system service NEW_SIGNAL, the kernel has another implementation of the same function, but is to be used to deal with signals sent from the environment. This service has some additional parameters. The "taskCode" parameter is used to specify which process, is to be used as an "interrupt handler", the next parameter is to specify which interrupt should be redirected. The third and the last parameter is used to specify if the process could be interrupted during execution or not, this is because some interrupts won't be cleared unless you've actually read a specific register.

```
SIGNAL NEW_INT_SIGNAL( PC taskCode, u08 interrupt, u08 Interrupted);

SIGNAL NEW_SIGNAL ( u16 number_of_args, ... /* arg1,arg2…*/ );
```

7.3.4 OUTPUT

SDL:

Output is used to transmit a signal from one process instance to another.
If destination is ambiguous, SDL has include the keyword VIA and TO to specify a unique receiver process instance, or the signal will be sent in a non-deterministic way; which is not tolerable in SDL description of a real system.
The first possibility to avoid this is to use output VIA; to specify on which signal route the signal is supposed to be sent, another possibility is to use output TO followed by the name of the receiver process. If more instances of the same receiver process exist, the output will have to use the receivers Pid to unambiguous determine which process instance which is supposed to receive the signal.

SDL REFLEX:

This is a complete SDL output function; the usage is on the other hand dissimilar. In SDL REFLEX *output* is implemented as a service with the same name, but it's usage is not exactly like SDL's description. This is because neither signal routes, nor channels have been implemented in SDL REFLEX, thus SDL REFLEX's output implementation only accepts the Pid (process identification) as a valid destination argument, this is always unambiguous, and thus will be able to handle all the previously mentioned scenarios. With the parameter signal_id, the signal id is to be specified, this is the value returned from the system service NEW_SIGNAL.
The dest parameter, is the Pid of the process which is to receive the signal.
If signal data is to be sent along with the signal, pointers to all the memory locations where the parameter data is located are to be inserted in the placeholders as follows. Signal data should neither be declared as static nor global.

void OUTPUT (SIGNAL signal_id, void *dest,...)

Figure 6 Output

7.3.5 TIME & TIMERS

SDL:

In SDL a timer is an object, owned by a process, that is able to generate a timer signal and put this signal into the input queue of the process. A timer can be activated with the *set* construct. The set construct has two arguments. First one is the absolute time for the expiration of the timer, and the other one is the name of the timer. For the specification of the expiration time, the expression *NOW* (of the predefined type *Time*, which is similar to *Real*) can be used. NOW always gives the current time during the interpretation of the system description. An activated timer can be deactivated with the reset construct. After resetting the timer, the process will behave in a way as if the timer never had been activated.

SDL REFLEX:

In SDL REFLEX no absolute timers are present, thus NOW doesn't exist in SDL REFLEX. In SDL REFLEX all defined timers are relative, thus if we had executed the statement SET(15,t), we would had to wait for an timer event in 15 timer units.
A timer unit in SDL REFLEX is by default set to 1ms.

In order to use SDL REFLEX timers, they have to be created. With the service NEW_TIMER() a new timer will be created, if memory is available; the function returns a TIMER object. To deal with the timer objects, SDL REFLEX has provided some operators:

- SET()
- RESET()
- ACTIVE()

The timer function ACTIVE() is used to check if a timer is counting(active) or not. Below we'll illustrate the effect of the SET and RESET on a timer, these statements are true for both SDL and SDL REFLEX.

Timer state / event	SET	RESET	Timeout
Stopped(inactive)	1. counting	2.stopped	
Counting	3.reset,restart counting	4.stopped	5.put timer into queue
Timer is in queue	6.remove timer from queue, restart counting	7.remove timer from queue, stopped	

Table 3.

7.3.5.1 NEW_TIMER

SDL:

This constructor is not implemented in SDL, but in a real implementation such a constructor has to be implemented.

SDL REFLEX:

This function creates a timer object if memory space is available. A timer is associated with its creator, and thus only the timer creator is allowed to do actions with the object.

```
TIMER New_TIMER ( void );                    // create a new timer.
```

7.3.5.2 SET

SDL/SDL REFLEX:

The function set the duration t for a timer, and to start the timer. If the timer is active when this function is executed, the timer will be reset. Thus the timer expiry is postponed with t timer units.

```
void SET ( u16 duration, TIMER timer );      // set duration to expire
```

7.3.5.3 RESET

SDL/SDL REFLEX:

This function is used to stop a timer. If a timer is active, but hasn't yet expired, the timer will be deactivated immediately. If the timer is inactive, but the signal which was sent previously still remains in the signal input queue, this signal will be removed from the queue.

```
void RESET ( TIMER timer_id );               // reset timer
```

7.3.5.4 ACTIVE

SDL/SDL REFLEX:

This function is used to examine if a specific timer is active, if true a boolean value is returned. If the timer is active 1 is returned else 0 is returned.

u08 ACTIVE (TIMER timer_id); // check if timer is counting

7.3.6 STOP

SDL:

After executing stop, the process instance and it's associated input queue and the signals are immediately destroyed. In that way the executing process instance becomes terminated.

SDL REFLEX:

This function is actually killing the process, and is usually associated with dynamically created processes. This call will release all memory occupied by the process. All signals in the signal input queue will be deleted, and all timers associated with the process are deleted (both active and inactive).
If the process to be killed has children, these children will stay alive but will be made orphans. This behaviour complies with SDL.

void STOP(void)

Figure 7 Stop

7.3.7 SAVE

SDL:

In SDL all signals reaching a process are stored into a FIFO (first in first out) queue associated to every process instance. The first signal ready to be consumed in the FIFO input queue of a process instance can be either input by the instance or saved, if no input symbol under the current state contains the signal name; or discarded, if the signal name is not specified in an input nor in a save. When a signal is saved, it remains in the input queue at the same position, and the next signal in the FIFO queue is examined to see if they can be input, saved or discarded.

SDL REFLEX:

This function does not really perform the assumed action described in SDL, but when used in association to WAIT_SIGNAL they comply with the SDL input concept. The SDL REFLEX service SAVE, is used to select a set of signals, which the process is going to search through when a signals arrives the process. At the end of the signal list, a special terminator should be applied, to inform the kernel of where to stop scanning for arguments. The select list is associated with the process, each time the process is to be resumed, the process scans through its input queue to see if any of the signals in the queue exist in the select list. If it does, this signal is to be consumed. If it doesn't the process checks to see if this signal exist in the save queue, if it does the signal remains in the input queue, and isn't discarded as it otherwise would.

void SAVE (SIGNAL signal,... /* sig1,sig2,…,sigN, END_LIST */);

Figure 8 Save

7.3.8 START

SDL:

In SDL, every process must contain exactly one start symbol. When a process instance is created, the first transition to execute is the transition beginning from the start symbol.

SDL REFLEX:

SDL REFLEX contains a function with the name *start*; this function is not intended to be used in any process. This function is only used to *start* the execution of the SDL REFLEX micro kernel, and thus can only be executed ones. *Start* is the last instruction to be executed in *main*.
The start function is initiating the environment, such as the tick timer, and finding first process to run. Load the process information, and then for eventually to start the process execution.

void START(void)

7.3.9 SELF

SDL/SDL REFLEX:

This function, when executed will return the Pid of the current process. The SDL REFLEX implementation of this functions is in accordance to SDL.

Pid SELF(void);

7.3.10 SENDER

SDL/SDL REFLEX:

This function, when executed will return the Pid of the process which sent the last consumed signal. The SDL REFLEX implementation of this functions is in accordance to SDL.

Pid SENDER(void);

7.3.11 PARENT

SDL/SDL REFLEX:

This function, when executed will return the Pid of the process which created the process executing. The SDL REFLEX implementation of this functions is in accordance to SDL.

Pid PARENT(void);

7.3.12 OFFSPRING

SDL/SDL REFLEX:

This function, when executed will return the Pid of the process of the last created child process of the executing process. The SDL REFLEX implementation of this functions is in accordance to SDL.

Pid OFFSPRING(void);

7.4 Additional SDL REFLEX system calls

In addition to the previously mentioned functions which are described in SDL, there are more functions in SDL REFLEX which isn't described in SDL, but is critical to completely implement the behaviour of SDL.

7.4.1 WAIT_SIGNAL

SDL:

All processes in SDL are either in transition or waiting. No symbol is specified to explicit describe a wait state, but obviously such a state is actually present.

SDL REFLEX:

This function has to be called each time a process is actually waiting for a signal to arrive, this function is used after INPUT or/and SAVE or/and SET.
It's absolutely necessary to perform this function call when a process is waiting for a signal, because this function puts the currently running process into sleep(blocked) until the pending signal is present. Meanwhile another process which is ready to run will be set to execute. Since, SDL REFLEX is to be used within a microcontroller which by nature only has one executing unit, all systems running within the controller does this in a pseudo-parallel manner, thus each process which is running, but should be sleeping is "stealing" valuable time from the other processes.

When this function eventually returns, it delivers the signal which arrived.
If the received signal contains data, these data could be read using the system call GET_SIGNAL_DATA this function is described in 9.4.2

SIGNAL WAIT_SIGNAL(void);

7.4.2 GET_SIGNAL_DATA

SDL:

Not described in SDL

SDL REFLEX:

This function collects all data connected to the last consumed signal; and transfers these data into the variables specified in the parameter list. All the parameter data types given as parameters to the parameter list have to match those parameters which were specified when the signal initially was created.

void GET_SIGNAL_DATA(void *param_1,... /* param_2,param_3 */);

7.5 Compatibility between specification and implementation

It seems like SDL is not much unlike the concurrent nature of the embedding environment, it is message oriented, the communication is asynchronous and each process has the behaviour of a state machine. In addition to these features, SDL has more to offer. Due to SDL's simple conceptual basis, SDL has maintained its original flavor, and is a formal language implying that every concept and construct in the language has a precise meaning and can be executed. With these facts it's should be reasonable to think that SDL would be an excellent choice for describing systems in embedded platforms. It's really an excellent choice, but when doing so, we have to take some issues into consideration.

Unfortunately, SDL and real time systems running within SDL REFLEX do not operate in same "environment". SDL is originally developed to describe distributed systems (telecommunication systems), where each SDL process runs independently on its own processor, unlike real time embedded systems running within an embedded platform, where all processes have to execute within the same processor, in a pseudo-parallel manner. In SDL, systems can be divided into independent subsystems that more easily can be designed, analyzed and composed. However in real time systems, executing in a pseudo-parallel manner, module independency is ruined by the fact that all modules running on one processor share the same time resource. Therefore after the composition of subsystems, the original real time properties of the individual subsystem can not be sustained. Another problem with real time systems running SDL REFLEX, is caused by the priority scheduling algorithm, is the possibility that processes with low priority are being indefinitely blocked, if the embedded systems is heavily loaded. This should never be the case in distributed systems where each process has its own processor. Priority aging could be used as scheduling algorithm to solve this problem, but this algorithm uses significantly more time to execute, and

much more program space is required to implement this behaviour, thus it's not suitable to implement this for a micro kernel operating system.

7.5.1 Timing constraints

In SDL, no transitions take time, neither does signal exchange through signal routes, thus when analysis is performed, timing is of no concern in SDL. However in a real time system, timing constraints are of very much importance. So a throughout performance analysis has to performed, after the SDL model is complete.

In SDL, modules can easily be added to a model without degrading the system performance. This is not the case in a real time embedded system, because each additional component would need timing resources to execute, thus the system would decrease its performance for each additional module applied; the result are that all timing analysis has to be redone for each applied system block.

7.6 Omitted SDL properties

During the development of the AVR microkernel, some SDL properties have been omitted. Down below, each of the omitted properties will be described in detail, and why these properties are left out in the kernel implementation.

7.6.1 Behaviour

7.6.1.1 Imported / Exported

In SDL a variable is only visible within the process in which it has been created. SDL has defined a simple way to interchange process data, by using the *imported* and *exported* value shorthand.

It's possible to exchange data without using *imported* and *exported*. If a process wants to access a variable value of another process, then a signal interchange with the process owning the variable could be arranged.

Since it's possible to achieve the same functionality with other constructs, we have decided to not implement the behaviour of *imported* /*exported* in the kernel implementation.

7.6.1.2 Service

The main advantage with *services* is that they can share common data. The disadvantage is that they are mutually exclusive (they share the same queue), and that they have to have disjunctive input signals sets (since the current signal consumed from the queue determines which service is to run). This way force unwanted renaming of signals just because you are using *services*.
One tends to prefer processes, unless one particularly wants to share data between the state machines, in which case *services* is one way of solving it. But not the only way; remote procedures, operations on data types or signaling to a "database" process are other ways of sharing data.
It is very complex to implement the service property in an RTOS. In our opinion it's not necessary. It's quite possible to achieve the same advantages with other SDL concepts. This concept has also been removed in the new SDL recommendation (SDL-2000) from ITU.

7.6.1.3 Continuous signal

SDL:

In describing systems, the situation may arise where a transition should be interpreted when a certain condition is fulfilled. A continuous signal interprets a Boolean expression and the associated transition is interpreted when the expression returns the predefined Boolean value true.

SDL REFLEX:

A continuous signal is an expression that is evaluated right after a process reaches a new state. It is evaluated before any message input or saved messages.
The behaviour of *continuous signal* can easily be achieved with conditional statements even if more *continuous signals* with distinct priority are present in the same SDL state. An IF…ELSE combination should be suitable as a substitution. Figure below shows an example.

```
SIGNAL SIGNAL_1,SIGNAL_2,SIGNAL_3;
void Process_A (void *params)
{
    enum {NOT_READY,READY} STATE = READY;
    SIGNAL receive;
    u08 N=2;
    for (;;)
    {
        switch(STATE)
        {
            case READY:
                if (N > 0)
                {
                    CREATE(Process_B,128,10);
                    OUTPUT(SIGNAL_3,OFFSPRING);
                    STATE = NOT_READY;
                }
                else
                {
                    INPUT(SIGNAL_1,SIGNAL_2,END_LIST);
                    receive = WAIT_SIGNAL;
                    if (receive == SIGNAL_1) {
                        /* Body 1 */
                    }
                    else {
                        /* Body 2 */
                    }
                }
                break;
            case NOT_READY:
                N = 0;
                STATE = READY;
                break;
        }
    }
}
```

Figure 9. Continuous signal code

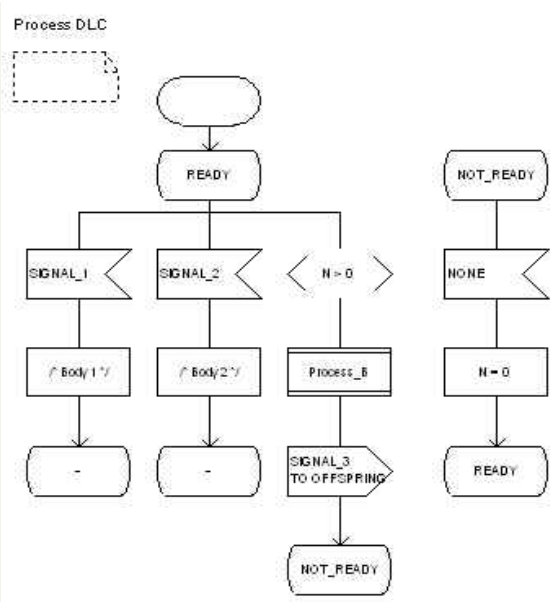

Figure 10. continuous signal SDL

7.6.1.4 Enabling condition

SDL:

An enabling condition makes it possible to impose an additional condition on the consumption of a signal, beyond its reception as well as on a spontaneous transition.

SDL REFLEX:

To describe this condition, two different scenarios might occur:

- The variable used as continuous signal is a local variable.
- The variable used as continuous is imported from another process.

To visualize these scenarios we've made two code snippets, one for each scenario. The first snippet uses a local variable.

```
void Process_C (void *params)
{
    enum {NEXT_STATE,READY} STATE = NEXT_STATE;
    SIGNAL receive;
    u08 N=1;
    for (;;)
    {
        switch(STATE)
        {
            case READY:
                INPUT(SIGNAL_1,END_LIST);
                WAIT_SIGNAL();                  /* Wait for the signal */
                if (N==0)                       /* The signal has arrived */
                {
                    STATE = NEXT_STATE;
                }
                else
                    /* Block and delete the process and all of its signals,
                    thereafter the kernel should be rescheduled.. */
                    STOP();
                break;
            case NEXT_STATE:
                INPUT(SIGNAL_2,SIGNAL_3,END_LIST);
                receive = WAIT_SIGNAL();
                if (receive == SIGNAL_2) N=1;
                else N=0;
                STATE = READY;
                break;
            case NOT_READY:
        }
    }
}
```

Figure 11 Enabling condition local variable.

```
void Process_C (void *params)
{
    enum {NEXT_STATE,READY} STATE = NEXT_STATE;
    SIGNAL receive;
    IMPORTED N;
    for (;;)
    {
        switch(STATE)
        {
            case READY:
                INPUT(SIGNAL_1,END_LIST);
                WAIT_SIGNAL();                  /* Wait for the signal */
                while (IMPORT(N)>0)             /* The signal has arrived */
                {
                    STATE = NEXT_STATE;
                }
                break;
            case NEXT_STATE:
                INPUT(SIGNAL_2,SIGNAL_3,END_LIST);
                receive = WAIT_SIGNAL();
                STATE = READY;
                break;
            case NOT_READY:
        }
    }
}
```

Figure 12 Enabling condition imported variable.

It should be obvious that this approach is troublesome for a multi tasking kernel. The problem is that the process enters a loop, and remains in this loop as long as N>0, thus losing valuable execution time. A much better approach would be to redesign the model. How this could be done is described in the figure below.

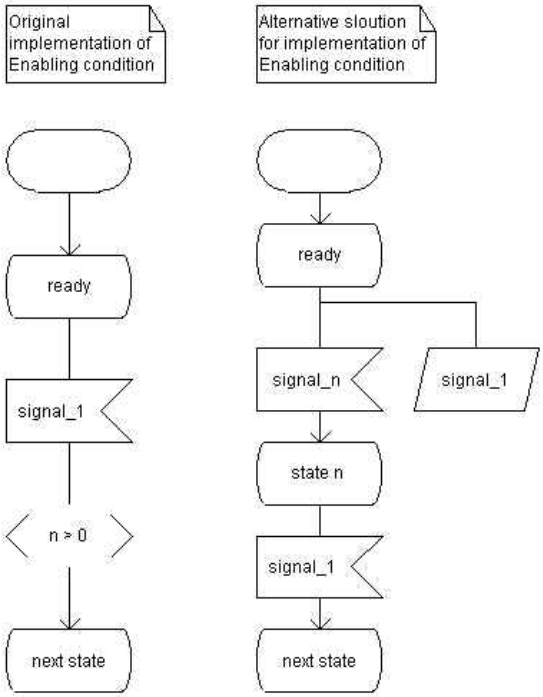

Figure 13 Enabling condition SDL

7.6.1.5 Spontaneous transitions

If the signal for an input is named NONE then the succeeding transition is a spontaneous transition i.e., the succeeding transition can be activated without any stimuli for the process. This will make the state-machine non-deterministic, in that way input 'NONE' will certainly not be used in an SDL description; input 'NONE' is mainly used for 'testware' parts of a model, such as protocol layers stubs. Thus, this will not be implemented in SDL REFLEX.

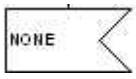

Figure 14 Spontaneous transitions.

7.6.1.6 View

As opposed to *import* / *export*, *view* always gives the current value of the revealed variable. This construct has been moved from SDL-2000, and is therefore not implemented in the real time kernel.

7.6.1.7 Optional transition string

The purpose of *optional transition string* is similar to optional definition, but transposed to transitions in a process. The symbol used is called an alternative. This is similar to #ifdef in ANSI-C

7.6.2 System Structure

SDL provides some entitles to structure a description into a hierarchy, in order to improve the system overview and for modularity; which is especially important in large and complex systems. The micro kernel real time operating system does not implement this hierarchy. This is because, the creation of such a hierarchy; will increase the kernel size considerably. The disadvantage is the lost ability to reuse code.

7.6.3 Communication

In SDL, the state machines contained in process instances communicate together or with the environment by transmitting and receiving signals through *channels* and *signal routes*.

The distinction between signal routes and *channels* are; signal routes communicate among processes, while *channels* communicate among blocks and the environment. All *channels* contain in addition a FIFO queue used to delay signals. Our real time operating system is designed to operate in a single chip solution. Within a SDL system where all processes executes on the same processor, neither channels nor signal-routes are necessary to create a connection between two processes.
Thus, both channels and signal-routes will be missing in the SDL REFLEX implementation.
All stimuli from the environment are handled by SDL REFLEX's internal interrupt handlers.

7.6.4 Constructs

To help designers when facing large descriptions, and to ease teamwork and maintenance, SDL provides some constructs. These constructs are:

- Package – To organize a description in several units
- Types of systems, of blocks, of processes and service – for better reusability
- Specialization (inheritance and virtuality)
- Context parameters

None of these constructs are implemented; these constructs are not essential to implement the behaviour of SDL, and is thus not implemented; though it would be nice to have, if more resources were available.

7.7 Data Types

Data in SDL are based on Abstract Data Types (ADT); which means that SDL doesn't implement dependent features such as the number of bits to store the different data types. An ADT is declared using the *NEWTYPE* construct, and is similar to a class in many programming languages; it provides a data structure plus some operations to manipulate the structure. The data structure can be enumerated values (*LITERALS*), a struct, an array, etc.

7.7.1 Predefined data

Predefined data types are defined in the Annex D to ITU-T recommendation Z.100. They are contained in the package called Predefined, implicitly used by an SDL description. The package Predefined also defines the operators which can be used on those types.
Most of the predefined data types, can easily be mapped to common ANSI-C primitives, their actual number of bits, is not defined in SDL but in a programming context, all primitives will have fixed data size. With matching operators, no work has been done, but in most cases the operators are same or equivalent for both SDL and ANSI-C.

Predefined SDL data types	C types	Byte Size
Boolean	Char	1 byte
Character	Char	1 byte
String	char*	2 byte
Charstring	char*	2 byte
Integer	Int	2 byte
Natural	unsigned int	2 byte
Real	Float	4 byte
Array	No equal !	
Powerset	No equal !	
Pid	Pid	2 byte
Duration, Time	unsigned long	4 byte

Table 4. Mapping of data types

7.8 Global time

The micro kernel real time operating system, has no absolute global time, all time is relative. In SDL the keyword NOW reference the actual global time. As a result, *NOW* isn't implemented. When we are setting *timers*, we'll set the *duration* rather then the exact time, when the *timer* is supposed to expire. In SDL a *timer* unit is not specified. Within the SDL REFLEX real time kernel, the time unit is by default set to 1us, but is configurable by the user.

8. Discussion

8.1 Introduction

The thesis motivation was to investigate the possibilities of using design automation software to develop software for embedded platforms and particular for the AVR micro controller family. To achieve this some research had to be done. As with all design automation software, they need a formal system description to capture the functional and non-functional requirements of the system. Hence, we started to investigate different formal description languages, and through thorough research we discovered different lacks of properties in most of these languages, thus these where to be rejected. Finally, we discovered SDL which seemed to be a worthy candidate for our mission. At first glance SDL seemed to perfectly satisfy all our needs. Hence, further investigation and analysis revealed some pitfalls that system designers should be aware of. Even after pitfalls where revealed, it seemed to be quite possible to use SDL to describe systems in embedded platforms. Through this research we were to discover the most prominent properties a real time operating system should possess, to succeed in describing the SDL behaviour. We choose to do a throughout research of the most common real time operating systems. The results from these investigations were not uplifting from our point of view, though some kernels fulfilled our demands. The disadvantage of these kernels was that they were not targeted for the AVR processor family, or if they were; they were to resource demanding. Thus, we started to create a framework for a kernel which purpose was to implement the most prominent properties of the SDL behaviour, and at the same time it should be able to fit into the AVR family of microprocessors. This task was indeed time consuming, but eventually we had made a micro kernel satisfying our needs. Though, the creation took some time, valuable discoveries where revealed with regard to how a SDL implantation has to be performed to achieve the expected results from a SDL system running within a multitasking environment.

8.2 Formal descriptive languages

Several tools, using different approach, exist on the market today. Formal descriptive languages, generally helps the designers express and verify their design ideas in an adequate way. This means that the language is expressive and unambiguous; it has platform-independent semantics, operational semantics and adequate support for modularization. Comparing the different approaches, tools available on the market today which use behavioural approach are preferred. This is due to the fact that almost complete code can be generated from a system description if state machines and state transitions describe the application. A behavioural approach system can be implemented with a RTOS designed for it since the behaviour of real time applications very well matches the behaviour of state machines.

8.3 SDL

SDL is a modeling language which primarily was developed for use in telecommunication systems including data communication, but as we've shown it can be used in all real time and interactive systems. It was designed for the specification and description of the behaviour of such system, i.e. the interwork between the system and its environment. It is also intended for the description of the internal structure of a system, so that the system can be developed and understood one part at a time. This feature is essential for distributed systems, and has also been widely used in the software industries for such systems. SDL covers different levels of abstraction, from a broad overview down to detailed design level. It was not intended to be an implementation language, but more or less automatic translation of SDL information to a programming language is, however possible and is especially used for developing distributed systems. However, for embedded systems, the situation is different, though it is possible to develop SDL systems if certain rules are followed.
Through our comparison among many formal description languages, SDL is the language which was chosen. Why is that ?
If we describe the nature of real time applications, you would be surprised to see how well this behaviour matches the behaviour of SDL.
SDL is especially suitable for control flow dominated systems, it's message oriented has a asynchronous communication and SDL processes behaves like state machine, thus this behaviour matches well the event driven nature of many real time applications. Hence, SDL became our number one choice.

8.4 Real time properties

To achieve the possibility of describing a SDL system in an embedded platform, the most prominent SDL features was to be implemented.
Much time was spent on discussions, on how to create a framework for such a kernel, though something was for certain, the kernel which was to be made had to emulate the behaviour of a true parallel executing system; this is actually impossible to achieve in an embedded system, but a pseudo-parallel system is in most cases suitable. Thus, we created a pre-emptive multitasking kernel; the kernel had to be pre-emptive to be able to adopt the behaviour of the SDL timer constructs, and to be able to react spontaneous on incoming signals as is the situation with interrupts. If these were the only criteria's for our kernel many real time kernels would be appropriate for our mission, but what's not that common for real time kernels, is how signal are to be handled.
Within a SDL system; a process should have an infinite input queue, where incoming signals are to be queued. When a signal has initiated a transition, it should be consumed and removed from the input queue. A process should also have the possibility to save signals for later use, if no input under the current state contains the received signal. A great effort was done, to implement this behaviour but without it, the kernel would never be able to describe any SDL systems.
Further, other SDL properties as triggers and transition elements where throughout analyzed to see if they were really necessary to describe systems. The result from this research is found in the chapter about "Omitted SDL properties" and in "Functional description of the SDL REFLEX microkernel".

8.5 Pitfalls

During our throughout research of SDL and through the kernel development process some pitfalls where discovered.
Some of these pitfalls where introduced because of our pseudo-parallel environment, some because of the SDL timer definitions, and some because of our resource constraint target. SDL which purpose is to describe processes running concurrently, has a property for describing continuous signals; such a construct is within an embedded platform inappropriate because it might be consuming a lot of valuable time, which should be used by other processes to run. Thus, the usage of this construct should be obeyed. SDL has some language aspects for expressing features of timed systems, these are unfortunately inadequate for hard real-time systems development, because they are indeterminable. When a timer is set to t seconds, the interpretation of this timer is in fact an arbitrary time duration dt (dt € [t,∞])). Such a weak interpretation of timers cannot provide enough expressive power to describe the timing behaviour of hard real time systems. Thus, the system should not be used for hard real time systems. The time mechanism in SDL is heavily affected by the platform-dependent physical clock. Such a platform dependent timing mechanism cannot provide facilities to debug and analyze timing behaviour of a model, because any debugging and analysis observation may introduce extra time passing, which changes the real-time behaviour of the model and leads to unreliable debugging and analysis results. Thus, whenever a change in a system description has been performed, all system analysis has to be redone.
In SDL there is no limit on how many processes that could be included in a system, neither are the concern of process size nor stack usage. However in a real time kernel operating systems, these parameters are of very much importance. Hence, these parameters has to be closely observed during the real time application development, if any of these parameters exceeds its size the system won't run or it will crash at some point. If its impossible to rewrite the system, to decrease t.ex code size the kernel should be ported to another processor. We've certainly made this possible because most of the kernel is written in the language of C, just a small amount of code is has to rewritten to move the kernel to another target platform.

8.6 Prototype

After the system was written and it worked correctly, bottleneck routines was identified and replaced with assembly-language routines. In this project GCC was used as development platform. GCC is open-source, free of charge and a port for the AVR-family exists. Most of the operating system was written in C to simplify maintenance and portability; however it was necessary to write some processor specific code in assembly. In addition to the properties already mentioned, GCC for AVR have lots of pre-built libraries, most of these libraries are reentrant, wich is requisite if they are supposed to collaborate with a pre-emptive operating system.

8.7 Future Work

SDL has some object oriented concepts that preferably could be added to SDL REFLEX, this hasn't been done because at present time there is no debugger for the AVR family that is able to debug code written in C++. But when eventually C++ is supported, SDL REFLEX should include support for these object oriented concepts. SDL is planned to be a profile of UML 2.0; so that executable unambiguous software can be produced. In this action some of SDL concepts is to be removed and in some is to be substituted by others which are which is not clearly defined yet, but when this work has been done, it would be of a great effort to rewrite SDL REFLEX to include these changes. To improve SDL REFLEX, we would like to invite other programmers to join for further development, thus SDL REFLEX will be distributed as open source through SOURCEFORGE under a GNU GPL (GNU Public Licence) license.

9. Conclusion

SDL has through this thesis been deeply investigated, and it seems like SDL is appropriate to use for most types of embedded systems, if a descent framework is offered that implements the simple semantics of SDL. If such a system is present, complex software systems could be developed in less time, and with less error than what's the case for conventional programming. Thus, the software development costs and time to market would be reduced. The matter of fact, to use this kind of software development could for some firms be the only way to survive in the business.

10. Bibliography

Silbergscatz, Abraham.
Galvin, Peter Baer.
Operating system concepts, fifth edition.

Doldi, Laurent.
UML 2 Illustrated.
November 2003

Comer, Douglas.
Operating System Design, the xinu approach.

Tanenbaum, Andrew S.
Woodhull, Albert S.
Operating system, second edition.
1997

Doldi, Laurent.
Validation of communication systems with SDL.
2003

Bræk, Rolv.
Haugen Øystein.
Engineering real time systems.
1993

Doldi, Laurent.
SDL Illustrated.
Mai 2001.

Labrosse, Jean J.
MicroC/OS-II, second edition.
2002

The ITU Telecommunication Standardization Sector (ITU-T)
www.itu.int/ITU-T/
ITU-T Recommendations.
Z.100 Specification and description language.
Z.109 SDL combined with UML.
Z.120 Message Sequence Chart (MSC).

Verschaeve, Kurt.
Combining UML and SDL.
System and Software Engineering Lab, Vrije Universiteit Brussel.

Mentor Graphics.
Embedded SoftwareWhite Paper.
Optimization Techniques for Risc Microprocessors.
Juli 1997.

OSE Systems.
OSE for AVR kernel, reference guide.
www.ose.com

Telelogic.
The Cmicro Library, reference guide.
www.telelogic.com

Progressive Resources LLC, Indianapolis.
PR_RTX reference guide.
http://www.prllc.com

SDL forum.
Tutorial SDL 88.
http://www.sdl-forum.org/sdl88tutorial/1/benefit.htm

What OS
Free Real Time Operating System (RTOS) solution
http://www.sticlete.com/whatos/

Free RTOS homepage
http://www.freertos.org/implementation/index.html

John Regehr's stack bounding page
http://www.cs.utah.edu/~regehr/stacktool/

11. Appendix

11.1 Users reference guide for SDL REFLEX.

11.2 Test case:

11.2.1 Lerret styring.

(CD-Rom)

11.2.2 Tappesystem

(CD-Rom)

11.3 SDL REFLEX source code.

(CD-Rom)

11.4 AVR datasheets

(CD-Rom)

11.5 WinAVR

(CD-Rom)

11.6 AVR studio 4.03

(CD-Rom)

11.7 Plugins for PN2

11.7.1 SDL Reflex syntax highlight

(CD-Rom)

11.7.2 Custom Tools

(CD-Rom)

11.8 Sample Makfiles

(CD-Rom)

11.9 GPL Gnu Public License policies

(CD-Rom)

11.10 Device drivers for the "HiA Trainer"

/CD-Rom)

11.11 SDL REFLEX – Users Reference Guide

(CD-Rom)