HIGH PRECISION AUTOMATIC SCHEDULING OF TASK SETS FOR MICROCONTROLLERS

By

Benjamin Ness

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

In partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2008

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Sandeep Neema

ACKNOWLEDGEMENT

This research was performed under the support of an NSF grant.

I would like to thank my advisor, Dr. Gabor Karsai for his invaluable support, advice, and instruction over the course of this project.

I also want to thank my wife, Hailey. Without her loving support and encouragement, it would have been much more difficult. Finally, I'd like to thank my parents. They never stopped encouraging me and inspiring me to perform to the best of my abilities and to set my goals high.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

RTOS	– Real Time Operating System
GME	– Generic Modeling Environment
UDM	– Universal Data Model
RM	– Rate Monotonic
EDF	– Earliest Deadline First

CHAPTER I

INTRODUCTION

Rapid development of applications for embedded systems is becoming increasingly important as the number and complexity of embedded devices increases. These applications vary widely in complexity, but typically have very tight constraints on timing. Developing these applications can be an extremely tedious task if performed by hand. This is especially true when the target microcontroller lacks the speed or resources to run a commercially available real-time operating system. In addition, there are few tools available to accelerate the development process.

Problem Statement

Designing applications for embedded systems presents a unique challenge. Because embedded systems are tightly coupled with their environment via sensors and actuators, precise timing is essential for proper function. In addition, microcontrollers are typically resource constrained. For a given application, it is not always possible to improve performance by upgrading to a faster processor. Lack of memory can not always be addressed by simply adding to the capacity. This is because there is not always a comparable processor available. Adding external memory may be possible, but power and space requirements may eliminate this option. Very small microcontrollers may not even have enough input and output connections to address external memory. And finally, power requirements can demand that the processor run in an even lower performance state than what it is capable of.

The combination of timing requirements and resource constraints may prevent a commercial RTOS (Real-time Operating System) from being used to schedule and manage the application. In addition, a RTOS may not be able to guarantee the correctness of its schedule at design time. In safety critical applications, such as the automotive or aerospace industry, guarantees of proper application function are critical. These applications may also have non-functional requirements, such as timing constraints. Often timing constraints are essential to proper operation, and guarantees on timing are needed.

When developing applications for embedded systems, the target architecture also plays a primary role in the design. Different architectures have different capabilities, and it is important to choose one that has the correct hardware for the required task. However, this complicates the developer's task. As systems evolve and change, so do the requirements. Although increased performance can be achieved through use of hardware level or assembly programming, the resulting code is very difficult to move from one platform to another. For this reason, it is essential that the code for the application remains in a portable form, with hardware-specific details abstracted away.

Requirements of a New Tool

One possible strategy is to design a completely new tool chain to expedite the development of real-time applications for embedded systems. There are a number of important elements to the usefulness and success of such a tool chain. These include a simple, easy to use environment, a powerful and efficient scheduler, and the ability to use for multiple target microcontrollers.

As with any application intended for use by the end-user community, ease of use and user friendliness are of high importance. A graphical interface provides a very effective means of describing the tasks and the relationship between tasks. This graphical representation is used to automatically generate a schedule.

The scheduler for the tool chain should be both powerful and efficient. It must be able to handle an adequate number of tasks as well as find a feasible schedule quickly. Because the tool's usefulness is dependent, in part, on its ability to offer guarantees about the correctness of its output, the accuracy of the scheduler is very important.

Finally, the output of the tool chain must be able to be used on any target microcontroller platform. Changing requirements may require that the application be moved to different target architectures. Making sure that the output of the tool is portable to a variety of target platforms is crucial to the tool's usefulness.

The Solution

Our objective was to develop a tool chain designed to expedite the development of deadline driven, real-time applications for microcontrollers. The input to the tool chain will be a set of real-time tasks supplied by the user. This task set will be represented in a domain-specific modeling language. GME (Generic Modeling Environment), a meta-programmable modeling toolkit [5], was used to design the modeling language. GME provides a flexible, easy to use graphical representation of the various elements of the modeling language. The user can create and modify task sets using GME. The modeling language allows the user to describe task sets composed of both periodic tasks as well as tasks driven by interrupt subroutines. The difference between the two is substantial, as interrupt driven tasks can not be controlled by the scheduling kernel. The modeling language also allows users to describe complex precedence relationships between periodic tasks. These constraints specify partial orders for task execution rather than dictating the actual execution order.

With the task sets generated by the user, the scheduler performs analysis on the input task set. This scheduler finds if a feasible schedule for the execution of the task set exists. Any feasible schedule is guaranteed to execute properly. Proper execution means several things. First, it means that no task deadlines will ever be missed. Second, no interrupt driven task should ever cause a periodic task to miss a deadline. These guarantees should hold so long as the worst-case execution times for each task, as provided by the user, are never exceeded.

Additionally, the tool chain also contains a simple, lightweight execution kernel. This kernel uses the generated schedule to execute the task sets correctly. The kernel allows the user to provide code for each task.

Once a feasible schedule for the task set has been determined, the kernel executes every task on time – that is, no deadline is ever be missed. Correct operation should be guaranteed so long as all the tasks stay within the bounds of the WCET (Worse Case Execution Time) as specified by the user at design time.

In the second chapter, we discuss the several different types of traditional scheduling techniques. We also look at two different tools that perform similar tasks. The third chapter details the specifics of the tool we designed. The fourth chapter describes the techniques used to analyze and test the tool. The fifth chapter summarizes the outcomes of the experiment. It also draws conclusions from the data gathered and considerations for future work on the tool.

CHAPTER II

BACKGROUND

Many real-time systems can be described by sets of periodically executing tasks. In real time systems, there are specific timing concerns that are related to both the start and the completion of task execution, as well as to the rates at which the tasks are to be executed. Additionally, there may be dependencies between these tasks. Each task performs some function of the whole system. In this thesis we assume that the worst-case execution time (WCET) of the tasks is known and finite.

There are two broad categories of scheduling techniques: offline and online [7]. Using offline scheduling techniques, the entire schedule is computed ahead of time. Offline scheduling algorithms require full knowledge of the task sets prior to execution. This is normally the case for embedded systems. Since the schedule is already determined, guarantees about timing and deadlines can be made. The same is not necessarily true for online scheduling algorithms. An online scheduling algorithm will choose which tasks to execute at runtime. Typically, online scheduling techniques introduce more computational overhead than offline methods. Additionally, because the online scheduler has no knowledge of how long tasks take to execute, or how frequently they need to execute, precise scheduling becomes extremely difficult. For these reasons, the scheduling method chosen for our tool chain is an offline algorithm.

Online scheduling algorithms are very flexible. Typically, they must make a scheduling decision each time a task is released or completed. Online scheduling algorithms are typically characterized by the method used to determine which task is chosen next. One of the simplest methods for online scheduling is the round robin method.

Under the round robin scheduling method, each task receives an equal share of the processor time [7]. This method is very easy to implement and works fairly well when the tasks have similar priority and required execution times. However, when task periods and execution times vary widely, the round robin scheduling method will often cause tasks to miss deadlines. This is because all tasks receive the same share of processor time without regard for how long those tasks actually take to execute.

Another online scheduling method is the rate monotonic (RM) method [7]. The simplest implementation of the RM scheduling method will always run the released task with the shortest period. In other words, priority is given to tasks that run at a higher rate. There are circumstances under which RM is an optimal scheduling algorithm. However, this normally depends on the use of preemption in scheduling. There is also an upper bound on the processor utilization for RM to still be optimal. Because of these restrictions, RM was not a feasible scheduling solution for our tool chain.

The Earliest Deadline First (EDF) online scheduling method is optimal scheduling method [7]. If the tasks can be executed without missing deadlines, EDF will schedule it correctly. EDF is only optimal on a preemptive, single processor implementation. If preemption is not possible, EDF is no longer optimal. In addition, task precedence constraints make it more difficult to issue guarantees about behavior and deadlines. Because the ability to guarantee accurate behavior and timing is essential for our application, neither RM nor EDF scheduling was suitable.

One of the first offline scheduling algorithms used to schedule periodic task sets on a single CPU is the cyclic executive [2]. A cyclic executive is typically implemented as a loop that contains subroutine calls with an optional delay at the end. Each subroutine call invokes a tasks' code. A cyclic executive, while simple, is completely deterministic in operation. That is, the tasks will execute in the same order, start at the same time, and complete at the same time for each execution of the loop. Any change to the task sets will require a recalculation of the valid schedule. However, cyclic executive methods may introduce excessive jitter in task release times and completion times. Each task in the frame allowed to execute as soon as the task before it completes. Any variance in the execution time of each task will introduce jitter in the following tasks. For our tool, we chose to use a time-triggered architecture. The release of each task is always related to a specific timing event rather than the tasks before or after it. In this way, jitter is reduced compared to a cyclic executive approach.

Xu and Parnas described a method for scheduling sets of real-time tasks on a single processing unit [10]. They described an algorithm to find optimal schedules for sets of real-time tasks with precedence and exclusion constraints on those tasks. This is an improvement upon the cyclic executive model, as it could be automated, while cyclic executives would typically be constructed by hand. In addition, it provides a method for dealing with preemption as well as precedence constraints. Each activity can be broken down into a set of segments which must be executed in sequence. In addition, each segment must be executed as a unit and cannot be preempted. The scheduling algorithm proposed here is a modified version of the Xu and Parnas algorithm. However, while the Xu and Parnas algorithm continue to explore the search space until it finds the optimal solution, the scheduler we used accepts the first valid solution it finds.

Giotto is a tool designed to bridge the gap between the actual control laws used in real-time applications and the software components that implement them [3]. Giotto is a coordination language. This means that it is used to describe the way in which the various computational tasks should be coordinated. In uses the concept of logical execution time, or LET. This allows each task to be executed at a specific time regardless of when the tasks before it complete. The elements of the Giotto language include tasks, ports, and modes. Giotto also includes the concepts of sensors and actuators. These represent software drivers for the hardware devices. It allows specific timing information to be added for each component, ensuring that writes to actuators and reads from sensors are performed at the correct time. Giotto compiles schedules for a single processor. These schedules are in the form of a high level language which is executed on the target platform. Our tool differs from Giotto in that sensors and actuators are not explicitly defined – they can be modeled as another task that handles the reads and writes. In addition, we allow the user to model tasks which are interrupt-driven.

The Timing Definition Language, or TDL, is conceptually modeled after Giotto [9]. TDL was designed to be a tool that could actually be used in the industry by developers. It is implemented as a high level language. Users specify the timing requirements for the various components of a time-triggered application using a textual syntax. It introduces features such as the ability to deploy onto multiple processors.

CHAPTER III

SOLUTION

Overview

This thesis presents a tool chain for deploying real-time applications onto embedded systems. The chain consists of three major components: the modeling language and its visual modeling tool, the scheduler / code generator, and the execution kernel.

Modeling Language

The modeling language was designed and implemented using the Generic Modeling Environment, GME [5] – a metaprogrammable visual modeling environment. The modeling language allows the user to describe the application in a natural, graphical way. This representation is independent of the platform on which it will be deployed. In this modeling language, the application is represented as a set of periodic real-time tasks, a set of interrupt driven tasks, and a set of precedence constraints on the periodic tasks. The main part of the metamodel for this language is shown in Figure 1. This metamodel, using the UML class diagram notation [6]. It captures the abstract syntax of the visual models of the language in a concise form.

A *PeriodicTask* represents any task that needs to be executed in a regular fashion. The attributes of the task allow the user to specify its period and worst-case execution time, both in microseconds. The deadline for each task is assumed to be same as the period.

A *PeriodicInterruptDrivenTask* represents any interrupt handler used in the application. These are assumed to be aperiodic, although an attribute is provided for the worst-case period, i.e. for the smallest period with which this task is invoked by the interrupt hardware. Again, a worst-case execution time attribute is provided.

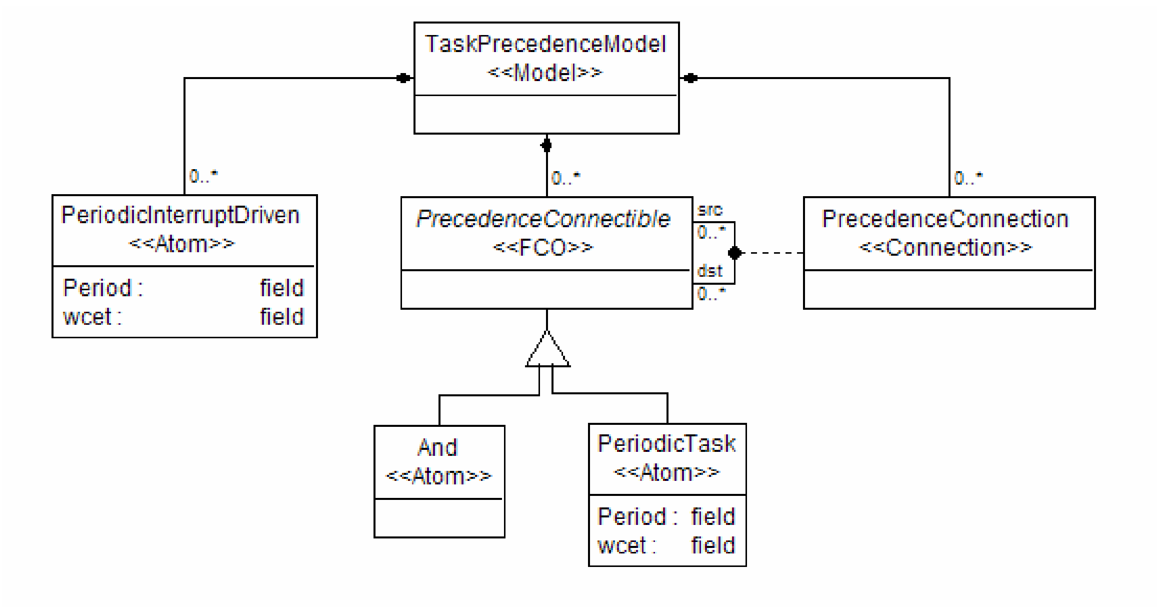


Figure 1 - Task Metamodel

Finally, *PrecedenceConnections* and *And* blocks are used to describe the relationship between *PeriodicTasks*. A *PrecedenceConnection* constraint between two *PeriodicTasks* implies that the source *PeriodicTask* must execute at least one time between executions of the destination *PeriodicTask*. If more than one *PeriodicTask* is connected using a precedence constraint, an OR relationship is assumed. That is, at least one of the source *PeriodicTasks* must be executed between each execution of the destination task. To describe an AND relationship between tasks, an AND block is used. A *PrecedenceConstraint* between two tasks can be described concisely in a mathematical expression. If *PeriodicTask* A must precede *PeriodicTask* B, and A(i) and B(i) are the start times of the ith job of *PeriodicTasks* A and B respectively, then

$$\forall A(i), \exists B(j) \quad st \quad A(i-1) < B(j) < A(j+1)$$

A similar expression can be written for every combination of *PrecedenceContraints* that can be drawn. All of the start times of every task that satisfies the Boolean function represented by the *PrecedenceConstraints* must fall between the start of a job and the start of the last job of the same task.

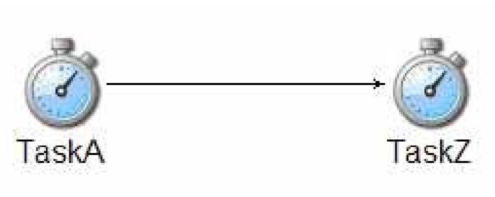


Figure 2 - A Simple *PrecedenceConstraint*

Figure 2 show a simple precedence constraint. In this example, TaskA must be executed at least one time between executions of TaskZ.

Figure 3 shows precedence constraints used in an “OR” fashion. Here, either TaskA or TaskB must execute at least once between executions of TaskZ.

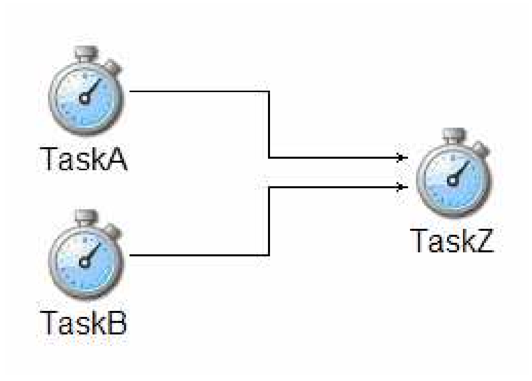


Figure 3 – A wired “OR” *PrecedenceConstraint*

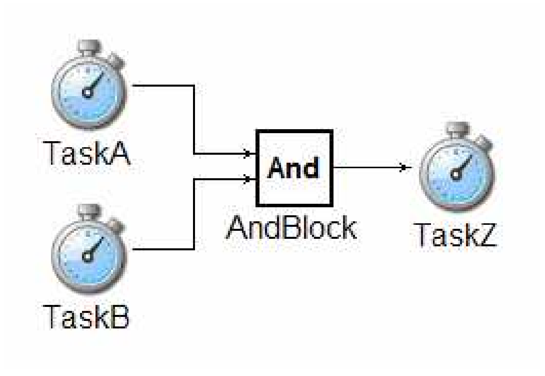


Figure 4 - "AND" *PrecedenceConstraint*

Figure 4 demonstrates the use of the *AndBlock*. In this figure, both TaskA and TaskB must execute between each subsequent execution of TaskZ.

Figure 5 shows a more complex arrangement of task precedence. Multiple *AndBlocks* and precedence constraints can be used to create constraints of arbitrary complexity. Any Boolean function can be constructed using the provided concepts. In this case, (TaskA AND TaskB) OR (TaskC AND TaskD) must be satisfied before TaskZ can execute.

Scheduler

The scheduler analyzes the task set to find a feasible execution schedule. The scheduler component is written as a GME interpreter using the UDM framework in C++ [1]. From within the GME environment, a single click on the toolbar will execute the scheduler component.

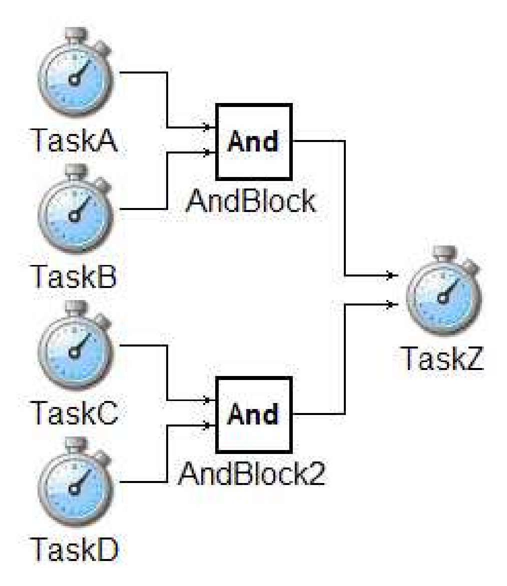


Figure 5 - Compound PrecedenceConstraint

For simplicity, the scheduling algorithm assumes non-preemption between all *PeriodicTasks*. Preemption adds additional complexity to both the scheduling algorithm and the execution kernel. Much more state information must be recorded as tasks execute, resulting in a much larger overhead than a non-preemptive method. However, there are some tasks sets which can be scheduled under a preemptive method, but cannot be scheduled non-preemptively.

In addition, it is assumed that any and all interrupt-driven tasks can and will interrupt each task execution with their worst-case period. The worst-case execution time of each task is adjusted upwards to correct for the effects of interrupts. An iterative, fix-point calculation is used to determine the maximum time that can be spent in interrupt routines during a task's scheduled execution time. This provides a safe, worst-case behavior prediction and helps ensure no deadlines will be missed.

```
Function CalculateNewWCET (TaskWCET, OldIntWCET)
        NewIntWCET = 0
        For Each InterruptDrivenTask i
                NewIntWCET += ceiling((TaskWCET + OldIntWCET) / i.Period) * i.WCET
        If (NewIntWCET != OldIntWCET)
                return CalculateNewWCET(BaseWCET, NewWCET)
        else Return TaskWCET + NewIntWCET
```

Pseudocode for Fix-Point Calculation of Task WCET

To find a feasible schedule, an exhaustive depth-first search is performed on the search space. At each step of the search, the scheduler will select the next task to be executed. Because no task can be preempted (and all worst-case execution times have been adjusted upwards for the interrupts), the scheduler has only two options at any point in time: Run an already released task, or wait for another one to be released. If any task misses a deadline, the scheduler removes that branch of the tree and backs up, trying a different decision. The process is repeated until a feasible schedule is found or the entire tree has been pruned, i.e., no feasible schedule exists. If no schedule exists, the user is informed and the scheduler exists. If a feasible schedule is found, the code generator executes.

```
Sort remaining jobs by deadline (EDF)
For each remaining Task t, Job j
        If (j meets deadline and j’s precedence constraints satisfied)
                Add j to end of schedule
                Schedule remaining jobs
                        If (Scheduled successfully)
                                Return Valid Schedule
        If (No valid schedules found)
                Return No Schedule
```

Psuedocode for scheduling algorithm

Because the scheduling algorithm is exponential in nature, there are potential problems with scaling as the number of tasks increases. However, typically, one of two scenarios is observed when executing the scheduler using real data. In the first, a solution is found very quickly. In the second, it takes a much longer time to arrive at a solution. We would like to make sure that most task sets fall into the first category, so that solutions are found very quickly. There is a “phase-transition” region where certain conditions will cause the solution to be exponential in nature. In our scheduler, some conditions identified are:

1. Very high total utilization
2. Large number of total jobs

However, neither of these conditions guarantees transition into the exponential region. Very high utilization task sets can still be scheduled relatively quickly most of the time. A large number of total jobs can occur even when the number of tasks is relatively low. If the periods of the tasks are not multiples of one another, the major cycle must be the least common multiple of the period of the tasks. This can cause the number of total jobs to be solved to be much higher if the major cycle is very large. When this condition is combined with a task set that is already difficult to schedule, the execution of the scheduler component typically takes much longer.

Code Generator

The code generator receives the output from the scheduler and generates a header file used by the execution kernel. This header file contains an array of function pointers, several arrays of integers, and constants used by the execution kernel.

The first part of the header is an array of function pointers. These point to the actual task functions, which must have the same name as the periodic tasks in the model (and consequently, must be valid C function names). This array is used to connect the actual task functions with the schedule.

The schedule is an array of integers, where each integer represents a task to be executed. This index is used to locate the task function within an array of function pointers. The schedule is cyclic - once all the tasks have been completed in order, the schedule starts again from the beginning.

The next array of integers is the start time for each task. The start times are actually the counts of the scheduler routine “ticks” that have occurred since the beginning of the current execution cycle.

The final array of integers contains the deadlines of each task. The deadline for each task also corresponds with the start of the next task. Because a feasible schedule is assumed, every task should complete before the next task could start.

Execution Kernel

The execution kernel is a simple, lightweight code written in C. It contains a main function and a scheduler function. The user must supply the code for each individual task, preferably in another include file.

The main function is a loop. At the beginning of the loop, the processor state is stored. This is used when a task runs past its deadline. Next, the function for the current task is called using the array of function pointers. If the processor is not currently scheduled to be executing a task, or when the function returns, the processor enters a low-power wait state until the scheduler interrupt occurs. After the interrupt, the loop restarts.

The scheduler routine is designed as a high-priority interrupt driven function. Many microcontrollers have a “Real-Time Interrupt” built-in. It can occur regularly and with high precision, both of which facilitate the function of the scheduler task. The main constant set for the scheduler determines the period at which the RTI fires. It can be between 1 microsecond and 1 millisecond, depending on the task set used.

On each execution of the scheduler routine, the “time” (an integer counter) is incremented. There are two cases that need to be analyzed: a task is currently executing, or no task is executing.

If a task is executing, there are two possible situations: the time is before the task’s deadline or the deadline has been passed. In the first case, the scheduler simply returns. In the second, the scheduler will set the current task to be the next task, load the saved processor state from the variables, and execute a return from interrupt instruction to break out of the task function and back into the main loop.

If no task is executing, the execution time of the next task is compared to the time. If it is time for the next task to begin, the current task value is adjusted, and the scheduler returns.

The code for the execution kernel was kept to a minimum in both space and time complexity in order to maximize available time for tasks. The code for the execution kernel can be found in Appendix A.

CHAPTER IV

EVALUATION

Method

Analysis of the scheduler was done using fairly simple task sets. The output of each task set was checked to verify that it is indeed schedulable.

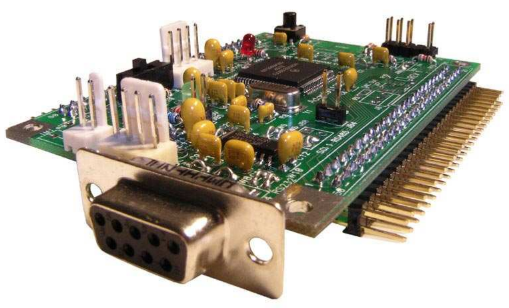


Figure 6 - Technological Arts AD9S12EQ128M0 Development Board

In order to analyze the generated code, the execution kernel was slightly modified. Several outputs were added to indicate the currently executing task, the "heartbeat" of the scheduler routine, and missed deadlines. Tasks were simulated using loops that took approximately the worst case execution time. The interrupt driven tasks were simulated using interrupts generated by the timer modules of the microcontroller. The code was then flashed onto a Freescale

MC9S12E128 microcontroller [8]. The microcontroller is on a Technological Arts AD9S12EQ128M0 development board, which has convenient output pins. A 34 channel Intronix LOGICPORT logic analyzer was connected to the outputs and used to monitor the execution of tasks [4].

Several different task sets were analyzed, from easily scheduled (low utilization) to barely schedulable (high utilization). It was verified that no tasks missed deadlines under these circumstances.

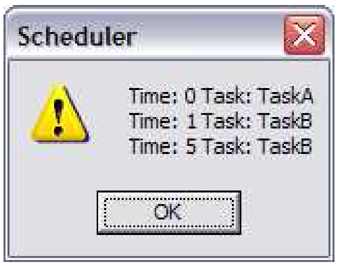
Finally, the frequency of interrupt driven tasks was increased beyond the given worst case values until tasks began to miss deadlines.

Example

To demonstrate how the tool works, we will walk through a few sample task sets. These task sets will have various levels of complexity. The first task set consists of two tasks. TaskA has a period of 10 ms and a WCET of 1 ms. TaskB has a period of 5 ms and a WCET of 2ms. The total utilization of this task set is 90%. There are no *PrecedenceConstraints*. The model for this task set is shown in Figure 7.

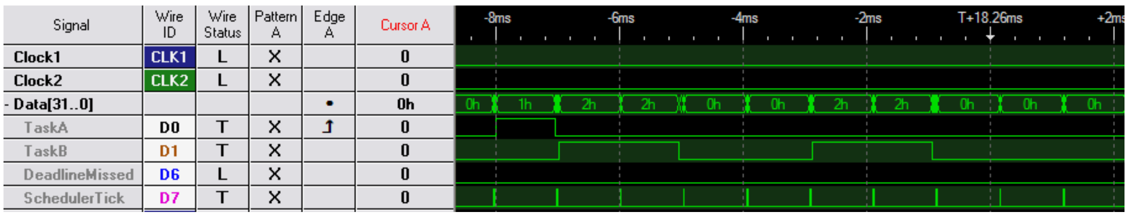
Figure 7 - Sample Task Set 1 Model

Although the utilization is fairly high at 90%, the task set is easily scheduled. The output of the scheduler is shown in Figure 8. Next, the header file is generated. The code is compiled and flashed onto the test board. Using the logic analyzer, a snapshot of the task set executing is obtained. One full major cycle of the set's execution is shown in Figure 9.



```
Time: 0 Task: TaskA
Time: 1 Task: TaskB
Time: 5 Task: TaskB
```

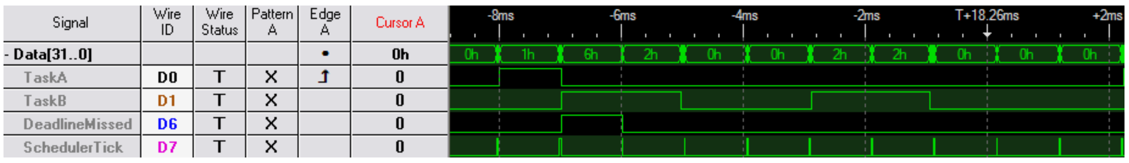
Figure 8 - Sample Task Set 1 Schedule



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
Clock1	CLK1	L	X		0
Clock2	CLK2	L	X		0
- Data[31..0]				•	0h
TaskA	D0	T	X	↥	0
TaskB	D1	T	X		0
DeadlineMissed	D6	L	X		0
SchedulerTick	D7	T	X		0

Figure 9 - Sample Task Set 1 Normal Execution

Looking at the logic analyzer output, it is observed that the "DeadlineMissed" output is never asserted. This means that the scheduler subroutine never observed a task running past its deadline. This case represents proper task execution. However, to demonstrate the flexibility of the execution kernel, we will introduce some task overruns to see how they are handled. For our first pass, we will cause TaskA to run longer than it should. The resulting output is shown in Figure 10.



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
- Data[31..0]				•	0h
TaskA	D0	T	X	↥	0
TaskB	D1	T	X		0
DeadlineMissed	D6	T	X		0
SchedulerTick	D7	T	X		0

Figure 10 - Sample Task Set 1 TaskA Overrun

In this run, the only clue that anything happened differently is that the "DeadlineMissed" output is asserted as the first job of TaskB releases. This indicates that job of TaskA had not completed execution when TaskB was scheduled to begin. Because of this, the scheduler used its routine to cancel the execution of TaskA and start executing TaskB instead. The "canceled" job of TaskA is considered failed and will not be allowed to complete execution at a later time. This strategy allows subsequent tasks to be released promptly and minimizes the chance that subsequent tasks will also miss their deadline.

The second task set consists of four tasks, precedence constraints on those tasks, and a single *PeriodicInterruptDriven* task. The tasks and their relevant parameters are shown in Table 1. The single interrupt has a worst-case frequency of 250 Hz, or 4 ms. It also has a worst-case execution time of 1 ms. The model, with precedence constraints, is shown in Figure 7.

Table 1 - Sample Task Set 2 Task Attributes

Task Name	Period	Worst-Case Execution Time
TaskA	6 ms	1 ms
TaskB	8 ms	1 ms
TaskC	12 ms	1 ms
TaskD	24 ms	2 ms

Once the model for the task set was created using GME, the scheduler was invoked via the graphical tool.

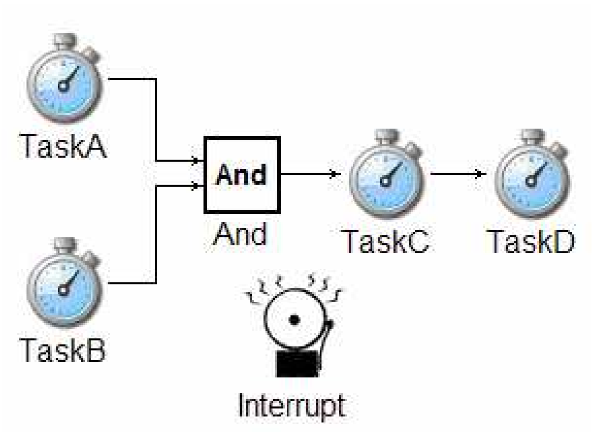
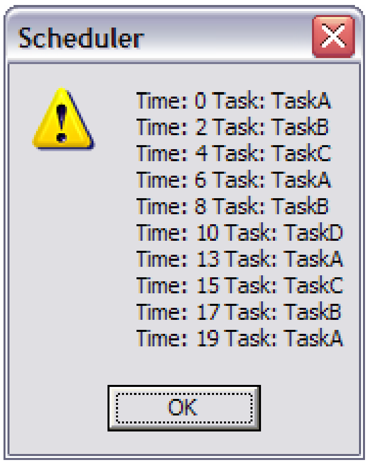


Figure 11- Sample Task Set 2 Model

The scheduler analyzes the precedence constraints and attributes and determines that this task set is indeed schedulable. The resulting schedule is shown in Figure 8. Finally, it prompts the user for the file name and path where the code generator should store the header file for the kernel.

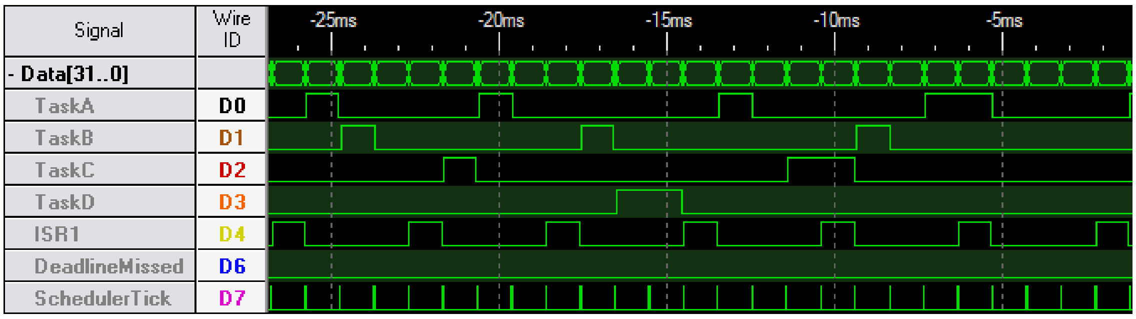


```
Time: 0 Task: TaskA
Time: 2 Task: TaskB
Time: 4 Task: TaskC
Time: 6 Task: TaskA
Time: 8 Task: TaskB
Time: 10 Task: TaskD
Time: 13 Task: TaskA
Time: 15 Task: TaskC
Time: 17 Task: TaskB
Time: 19 Task: TaskA
```

Figure 12 – Sample Task Set 2 Schedule

The last step in the process is to compile the kernel with the new header file and flash it onto the microcontroller. Once this is done, we can use the logic analyzer's software to take a sample of the outputs displaying the current system state. One full hyperperiod of the schedule, 24 ms, is shown in Figure 9.

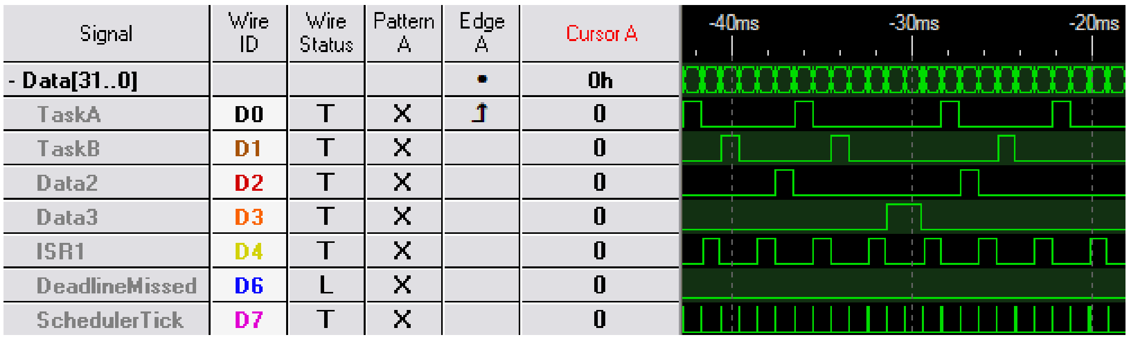
Because the scheduler interrupt will assert the "DeadlineMissed" output whenever a task has run past its deadline, we can quickly check whether or not deadline misses occurred. In this figure, the "DeadlineMissed" signal stays low during the entire hyperperiod. A more careful check reveals that every task is actually completed before its deadline.



Signal	Wire ID
- Data[31..0]	
TaskA	D0
TaskB	D1
TaskC	D2
TaskD	D3
ISR1	D4
DeadlineMissed	D6
SchedulerTick	D7

Figure 13 – Sample Task Set 2 Normal Schedule Execution

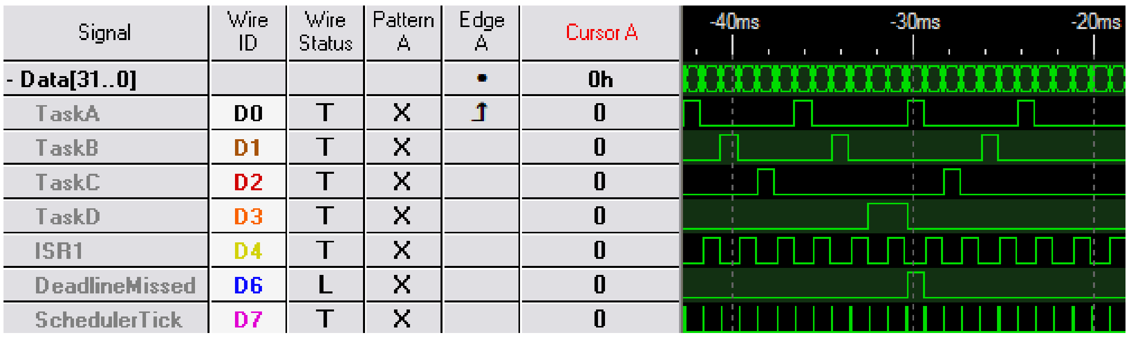
We will now demonstrate what happens when ISR1 is allowed to execute faster than the maximum rate under which the schedule was generated. Originally, it was limited to a maximum 250 Hz rate. Increasing this maximum rate has the potential of causing other tasks to miss their deadlines. Because the scheduler algorithm has no control over the rate of incoming interrupts, increasing the maximum frequency introduces the possibility that *PeriodicTasks* will miss their deadlines. Figure 14 shows the resulting output when the worst case rate is increased to 333 Hz.



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
- Data[31..0]				•	0h
TaskA	D0	T	X	↥	0
TaskB	D1	T	X		0
Data2	D2	T	X		0
Data3	D3	T	X		0
ISR1	D4	T	X		0
DeadlineMissed	D6	L	X		0
SchedulerTick	D7	T	X		0

Figure 14 - Sample Task Set 2 ISR1 333Hz

Even with ISR1 executing at a maximum rate of 333 Hz, no task misses a deadline. Going back to the model and executing the scheduler reveals that this change does not make the task set “unschedulable”. We will further increase the maximum rate at which ISR1 fires to 500 Hz. The results are shown in Figure 15.



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
- Data[31..0]				•	0h
TaskA	D0	T	X	↥	0
TaskB	D1	T	X		0
TaskC	D2	T	X		0
TaskD	D3	T	X		0
ISR1	D4	T	X		0
DeadlineMissed	D6	L	X		0
SchedulerTick	D7	T	X		0

Figure 15 - Sample Set 2 ISR1 500Hz

With ISR1 firing at 500 Hz, TaskD begins to miss deadlines. However, all other tasks continue to meet their deadlines. In this case, backing out of tasks that fail to meet deadlines allows other tasks to still execute promptly.

CHAPTER V

CONCLUSION AND FUTURE WORK

Conclusion

The testing revealed that every task set for which the scheduler component produced a valid schedule never missed a deadline, even when subjected to the worst-case execution times for each task. As the interrupts were added and increased to occur with their maximum frequency, it was observed that deadlines were still met. Even task sets above 80% utilization still executed as expected.

In addition, when tasks were allowed to run past their deadline, the next tick of the scheduler task identifies the overrun and performs its routine to back out of the past-due task. It then executes the next task in the schedule.

It was also observed that overhead of the scheduler task is approximately 15 microseconds per execution. For task sets requiring millisecond accuracy (up to 1 kHz), the total overhead is less than two percent.

In this particular implementation, the scheduler period was selected as 1.024ms. The prescaler value for the real-time interrupt was chosen accordingly. Analysis of the scheduler ticks shows a typical scheduler period of 1.0253ms, or within .001% of nominal. Also, the release time jitter was evaluated. Tasks are typically released within 0.1ms of the nominal release time per the schedule.

In conclusion, the tool chain developed greatly simplifies the process of developing real-time applications. The front-end modeling language allows the user to express a wide variety of task sets with strict timing and ordering requirements. The scheduler processes the data from the model in an efficient manner, finding a feasible schedule if one exists. The execution kernel provides a lightweight, low overhead means of executing the tasks according to the generated schedule. It has been experimentally shown to have very low jitter between task release times.

Based on experimental data, these schedules can be executed with a high degree of confidence that timing data is correct and that deadlines will not be missed.

Future Work

Although the tool chain is useful in its current form, there are a lot of improvements that could be made. Additions of such task attributes as relative deadline or phases would make the modeling language much more flexible in the applications it can describe. In addition, the scheduler could use optimizations to find feasible schedules faster.

The scheduling algorithm, although exhaustive, settles on the first feasible schedule it locates. A more useful schedule would be the ideal one, which minimizes latency along the some path. It would be very desirable to minimize latency along the worst-case path. The worst case path would be one with the longest sequence of data dependencies or precedence constraints. The Task priorities could also be used to enhance the scheduler.

Finally, the execution kernel is currently platform dependent. A more useful implementation could allow the user to implement the platform specific functions in another file. In this way, the generated code could be used on a wide range of microcontrollers rather than only the Freescale MC9S12E128.

APPENDIX A

SOURCE CODE FOR EXECUTION KERNEL

```
#include <stdio.h>
#include <string.h>

#include "mc9s12e_regs.h"
#include "tasks.h"
#include "myheader.h"

int main(void);

void scheduler(void) __attribute((interrupt));

volatile short int currenttask = 0;
short int n = 0;
unsigned short int taskstarttime = 0;

int statusSP = 0;
int statusD = 0;
int statusX = 0;
int statusY = 0;
int statusC = 0;


volatile int time = 0;

int counter = 0;

int main() {
    CTCTL.byte = 0x08; /* disable COP watchdog timer */
    RTICTL.byte = prescale;

    CRGFLG.byte |= RTIF;
    CRGINT.byte |= RTIE;  //Enable RTI

//Configure timer module for interrupts

    TIM0.TIOS.byte |= IOS6;     //Channel 6 used for Output Compare
    TIM0.TSCR2.byte |= 0x00;    //Prescale factor of 1
    TIM0.TIE.byte |= C6I;       //Use interrupts for channel 6
    TIM0.TFLG1.byte |= C6F;    //Clear interrupt flags for channel 6

    TIM0.TSCR1.byte |= TEN;    //Enable Interrupt Module

    while (1) {

          //Store register status

          asm("SEI");             //Disable Interrupts

          asm("STD statusD");     //Store D
          asm("STX statusX");     //Store X
          asm("STY statusY");     //Store Y

          asm("PSHD");
          asm("TFR CCR, D");
          asm("STD statusC");     //Store CCR
          asm("TFR SP, D");
          asm("STD statusSP");    //Store SP
          asm("PULD");

          asm("CLI");             //Enable Interrupts

          //Label to jump to if deadline missed

          asm("MainLoop:");
          if (currenttask >= 0) {
             //Call current task using array of function pointers
             funcArray[currenttask]();

             //Task finished!
             currenttask = -1;
             n++;
          }

          //Wait for scheduler interrupt to fire
          asm("WAI");
    }

    return 0; /* not used */

}
```

```
void scheduler() {
     CRGFLG.byte |= RTIF;      //Clear interrupt flag

     int jumpout = 0;
     time++;

     //3 Cases where action is needed:

     //1: End of hyperperiod,
     //2: No task is running and one needs to be released,
     //3: Task past deadline

     if (time >= hyperperiod) {
        time = 0;
        n = 0;
        if (currenttask != -1) jumpout = 1;
        currenttask = schedule[0];
     } else if (currenttask == -1) {
         if (time >= scheduletime[n]) currenttask = schedule[n];
     } else

      //Check for missed deadlines, back out of current task

      if (time >= deadline[n]) {
             n++;
             if (time >= scheduletime[n]) currenttask = schedule[n];
             jumpout = 1;
      }

     if (jumpout == 1) {
             /*When a deadline is missed, we want to cancel the current
             task and return to the top of the task loop. This is done
             by zeroing the stack and performing an RTI back to the main
             loop. */

             asm("LDS statusSP");

             asm("LDD #MainLoop");
             asm("pshd");

             asm("LDD statusY");
             asm("pshd");

             asm("LDD statusX");
```

```
                asm("pshd");

                asm("LDD statusD");
                asm("pshd");

                asm("LDAA statusC");
                asm("psha");

                asm("RTI");
        }

}
```

REFERENCES

1. Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A.: The Design of a Language for Model Transformations, Journal of Software and System Modeling, 2005.

2. T. P. Baker, and A. Shaw, "The cyclic executive model and Ada," Journal of Real- Time Systems, vol. 1, June 1989.

3. Henzinger, T.A., Horowitz, B. and, Kirsch, C.M. Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE. Volume 91, Issue 1, Jan. 2003 Page(s):84 – 99

4. *Intronix LA1034 LogicPort PC-Based Logic Analyzer with USB Interface*. Retrieved March 21, 2008. Web site: http://www.pctestinstruments.com

5. Karsai,G.: “A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming”, IEEE Computer, V.28. pp. 36-44., March 1995.

6. Pilone, Dan. (2003) UML Pocket Reference. Sebastopol, CA: O’Reilly Media Inc.

7. Shaw, Alan. (2001) Real-Time Systems and Software. Hoboken, NJ: Wiley.

8. *Technological Arts*. Retrieved March 21, 2008. Web site: http://www.technologicalarts.ca/catalog/index.php?cPath=50_52

9. Templ, J. TDL Specification and Report. Technical Report, University of Salzburg, Austria. Mar 2004, www.SoftwareResearch.net/site/publications/Co59.pdf

10. Xu, J. and Parnas, D.L. 1990 Scheduling Processes with release times, deadlines, precedence and exclusion relations. IEEE Transactions on Software Engineering, Volume 16, Issue 3. Page(s): 360-369