Implementation and Analysis of a Real-Time Scientific Payload Control System

Due on 5/7/2018 at 3:10pm

Dr.Albert Cheng

Andrew G. Walker

Contents

Introduction **3**

Design **4**

Hardware 4

FreeRTOS 5

Overview 5

Tick System 5

Model Checking **5**

Proving Correctness Using NuSMV 7

Implementation **8**

Designing a Feasible Task Set 8

Determining WCET 8

Schedulability Analysis 9

Leveraging the Idle Task for Saving Power 10

Power Saving 11

Conclusion **12**

Appendix A: Formal Specification in SMV **14**

Introduction

Each year, NASA and the Louisiana Space Consortium graciously host payloads through the High Altitude Student Payload (HASP) program from students around the world on a high altitude balloon flight as a test bed for scientific and engineering projects. The harsh conditions in the stratosphere provide interesting engineering challenges and necessitate meticulous design and planning. Payloads must be designed to operate in near vacuum conditions with temperature fluctuations of up to 80 °C. Additionally, payloads must be designed to operate on less than 60 W of power and have the ability to operate autonomously in case of a communication failure.

The Stratospheric Organism and Radiation Analyzer (SORA) payload from the University of Houston was accepted for the 2018 flight and should fly sometime in August. The purpose of this project was to apply techniques from real-time systems including scheduling and model checking to specify and implement a real system. In this report, the details of the specification and design of the SORA payload's control systems are presented as well as an analysis of the system's overall effectiveness.

Figure 1: HASP gondola in the stratosphere.

Design

The SORA payloads main purpose is to sample for microorganisms present in the stratosphere using two modified diaphragm vacuum pumps and solenoids that can be opened and closed to seal collection chambers. The system will have 8 temperature sensors and two barometers to monitor the environmental conditions of the payload and the temperature of the pump system. The control system must be capable of monitoring environmental conditions and the health of the two pumps while accepting commands from ground control. It must also be able to ensure that if the conditions become unsuitable for collection or if the pumps become overheated the collection system is powered off and cannot be restarted until conditions return to normal. Specific constraints are listed below.

- Collection has to occur at or above 26 km
- Both pumps must maintain a temperature between 0 °C and 50 °C
- Commands to start collection will only be accepted if the pumps are healthy and the payload is at the correct altitude
- Commands to stop collection can be accepted at any time the system is in collection mode
- The system should be able to respond to commands within 50 ms
- If either the altitude is too low or the pumps are not at a healthy operating temperature the collection must be stopped and an error is raised

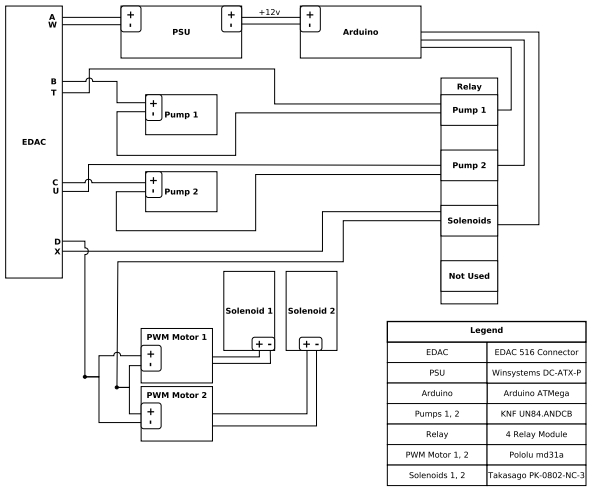

Figure 2: Simple block diagram showing system design.

Hardware

The core hardware utilized for this project is the Arduino Mega 2560 R3, which is a development board base around the ATmega2560 microprocessor. The Arduino is a robust development board that provides 54 digital input and output pins, 16 analog pins, I2C interface and 4 separate hardware serial ports for interacting with various sensors and auxiliary components. The Atmega2560 operates at a clock rate of 16 MHz and is used widely used for low power micro-controller applications.

An auxiliary board that sits on top of the main Arduino board will be utilized log sensor data and system messages. A relay board controlled by digital output pins is also utilized turn the pumps on and off. Commands from ground control are accepted via discrete commands and are first processed by digital logic and then fed into the Arduino as a simple low or high signal indicating whether a command is ready to be processed.

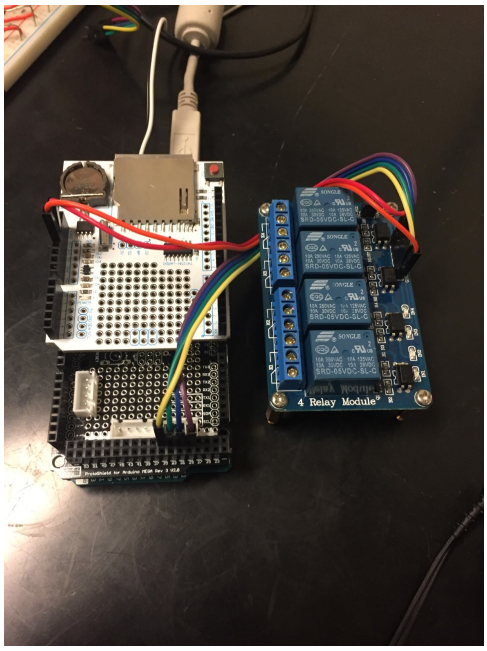

Figure 3: Arduino and accompanying data collection shield(left) and relay board(right).

FreeRTOS

Unfortunately, the Atmega2560 and other processors in the AVR family are not supported by VxWorks as of yet. However, FreeRTOS is an open source real-time operating system that has been ported to a wide variety of hardware including the AVR family of microprocessors. In terms of functionality, FreeRTOS provides nearly all of the same features that VxWorks does, including priority based and round robin scheduling, built in support for semaphores and queues, hook facilities, and a robust faculty for tracing task execution.

Overview

FreeRTOS uses a static priority based scheduling algorithm that at every time instance selects the task with the highest priority to run. Other scheduling algorithms such as EDF and LLF have been implemented by various programmers in the open source community however they are not available by default and require an implementation for whatever specific hardware the system is running on.

Tick System

In any real time operating system it is imperative that the highest priority ready task be executing at any given point in time. Due to the discrete nature of microprocessors this is generally implemented as a system

interrupt that happens at fixed intervals, wherein the scheduler interrupts the running task and determines whether to continue execution of that task or switch to a higher priority ready task. Specifically for the Arduino port of FreeRTOS, the on board watchdog timer is utilized to generate an interrupt at a user defined interval ranging from every 15 ms up to every 500 ms. Choosing the tick interrupt interval generally comes down to how responsive the system needs to be and how much time the processor will be idle. Short tick interrupts provide very fast system response for applications in which many different tasks will be preempting each other and a longer tick interrupt can be useful when sending the processor to sleep for long periods during CPU idle time.

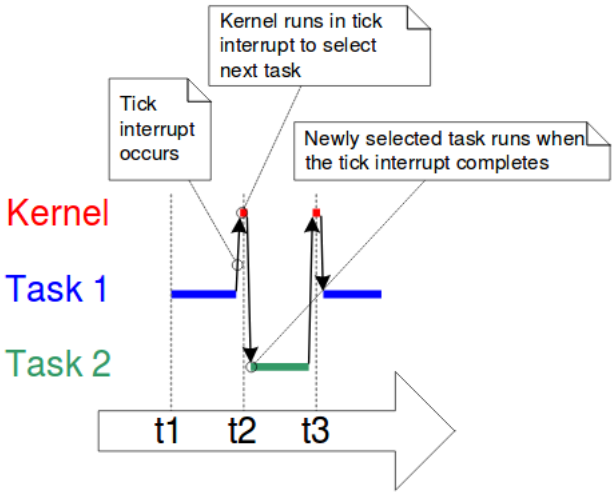

Figure 4: Demonstration of the kernel tick interrupt.

Model Checking

A classical approach to modeling concurrent systems originally described by Clark et.al [1] is to specify them as a class of finite state automata referred to as a Kripke Structure. A Kripke Structure is composed of a set of states, atomic propositions true in each state and a set of binary relations between each state. One can then use a type of temporal logic called Computational Tree Logic (CTL) to prove various properties of the specified model. The Kripke structure corresponding to the SORA payloads specification and the propositions true in each state are shown in Figure 5 and Table 1 respectively.

State	Pumps Running	Solenoids Engaged	Start CMD Received	Stop CMD Received	System Healthy	Error
S_0	False	False	False	False	False	False
S_1	False	False	False	False	True	False
S_2	False	False	True	False	True	False
S_3	False	True	True	False	True	False
S_4	True	True	False	False	True	False
S_5	False	True	False	False	False	True
S_6	False	False	False	False	False	True
S_7	False	True	False	True	True	False

Table 1: Table of atomic propositions true in each state.

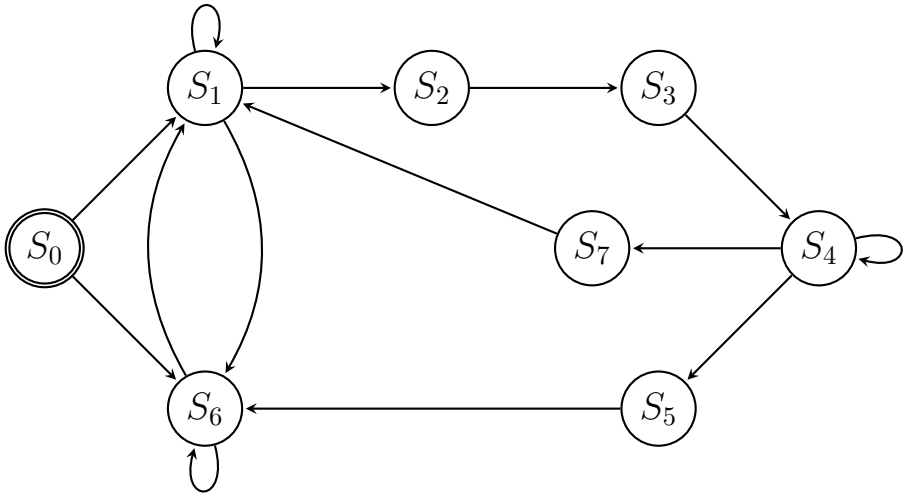

Figure 5: Representation of system specification as a Finite State Automata

CTL can specify a set of logical constraints that must be satisfied by the the model. After defining the model in a modeling language a model checking tool can be used to explore the state graph and ensure that the CTL formula is satisfied. Using the SMV language [2] developed at Carnegie Mellon University, I was able to model the SORA control system and model check its safety assertions specified in CTL defined in Table 2. A full description of the system in the SMV language is provided in Appendix A.

CTL Formula	**English translation**
$AG(\neg e \cup \neg b)$	Pumps will never be running when there's an error.
$AG(d \rightarrow AF(\neg a \wedge \neg b))$	When the stop command is received the pumps and solenoids eventually turn off.
$AG(\neg e \wedge a \rightarrow AF(\neg a \wedge \neg b))$	When the system becomes unhealthy while running the pumps and solenoids eventually turn off.
$AG(c \wedge e \rightarrow AF(a \wedge b))$	When the start command is received and the system is healthy the pumps and solenoids eventually turn on.
$AG(e \rightarrow AF(\neg a \wedge \neg b))$	When the system becomes unhealthy the pumps and solenoids eventually turn off.

Table 2: CTL formulas and their English translations.Note that the symbols a,b,c etc. correspond to the propositions defined in Table 1 e.g a = "Pumps Running" b = "Solenoids Engaged" etc.

Proving Correctness Using NuSMV

Using the reimplementation of the SMV model checker NuSMV [2], I was able to verify that the CTL specifications are satisfied by the model created in SMV. The output of the model checker, shown in Figure 6, shows that all of the CTL specifications from Table 2 are true. An interesting feature of this model checker is that given a CTL specification that is not true, the model checker will provide a counter-example showing a series of state transitions that lead to a case where the formula is false. From the output one can see that the assertion which in English corresponds to, "If the start command is received all paths of execution

eventually result in the pumps turning on" is determined to be false. The output then shows that in the case where the start command is received and the system is not healthy, the pumps will not be turned on and an error will be raised. A full graph representing all transition relations is shown in Figure 7

```
-- specification AG (system = unhealthy -> AX error)  is true
-- specification AG (cmd = start -> AF pumps = on)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    system = idle
    pumps = off
    solenoids = disengaged
    cmd = none
    error = FALSE
  -> State: 1.2 <-
    system = unhealthy
    cmd = start
  -- Loop starts here
  -> State: 1.3 <-
    error = TRUE
  -> State: 1.4 <-
-- specification AG (cmd = stop -> AF pumps = off)  is true
-- specification AG ((cmd = start & system != unhealthy) -> AF (pumps = on & solenoids = engaged))  is true
-- specification AG ((cmd = stop & system != unhealthy) -> AF (pumps = off & solenoids = disengaged))  is true
-- specification AG (system = unhealthy -> AF (pumps = off & solenoids = disengaged))  is true
-- specification AG (!error | pumps = off)  is true
```

Figure 6: Output of NuSMV specification verification

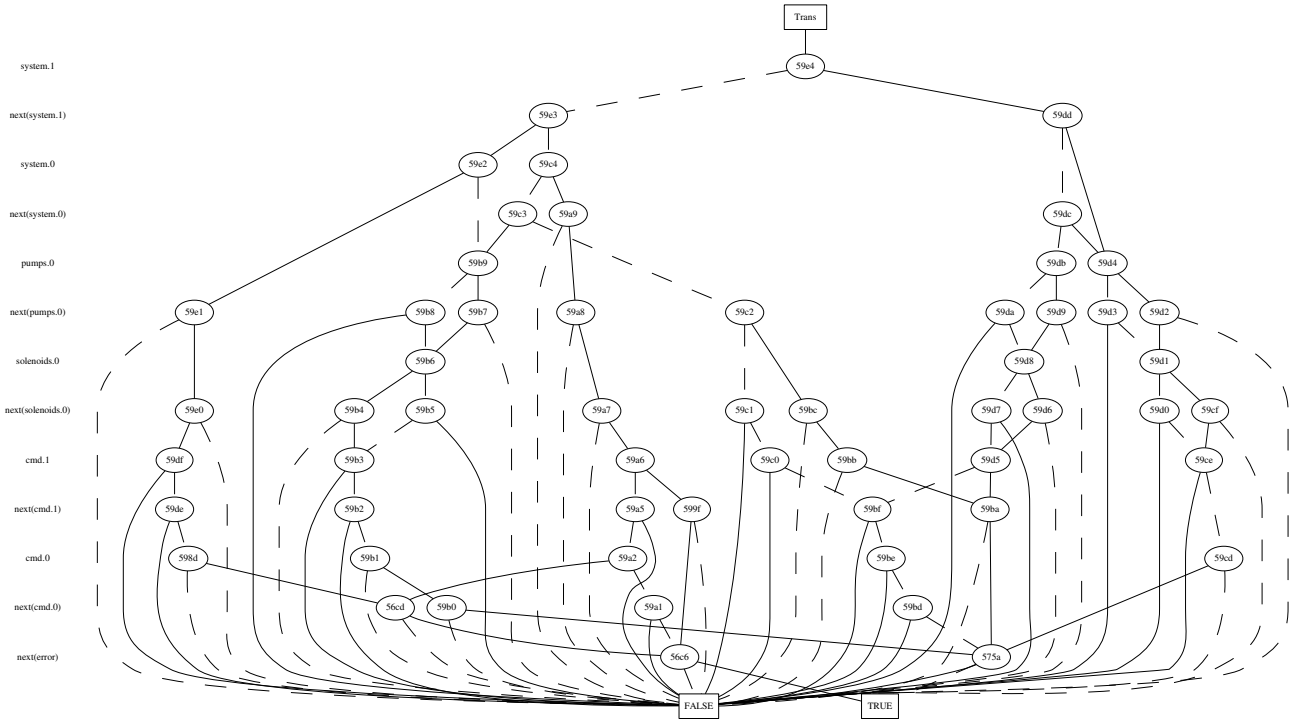

Figure 7: Graph generated by NuSMV that displays all transition relations.

Implementation

Designing a Feasible Task Set

The task set is fairly simple and is composed of periodic and sporadic tasks. Tasks that monitor the temperature and pressure of the payload are modeled as periodic tasks that are executed every 25 ms and 30 ms respectively. Tasks that monitor the health of the payload and check for new commands are modeled similarly with a period of 50 ms. Tasks that are performed in response to a command being received are modeled as sporadic tasks that can occur with a minimum separation of one period of the command checking task. The execution time of these sporadic tasks is accounted for in worst case execution time of the command monitoring task. Determining if the task set can be feasibly scheduled given the desired system response time of 50 ms requires a priori knowledge of tasks start time, periods, and worst case execution time (WCET).

Determining WCET

Determining the WCET of a task is not trivial and there are a wide variety of techniques for doing so. In general, techniques fall into two categories: measurement of the task execution times empirically or static analysis of a tasks actual instructions. For this project, I decided to use a simple measurement method. The Arduino has a built in timer accurate to 4 µs that was utilized in combination with kernel hooks provided by FreeRTOS to time the tasks while the system was in operation. The best case, worst case, average case and standard deviation of various for various system actions is shown in Table3 and each tasks execution time in microseconds and period is displayed in Table 4.

System Action	**Best Case**	**Worst Case**	**Average Case**	σ
Temp. Read	476	528	504.31	3.44
Press. Read	676	712	694.28	2.87
Punch Solenoid	16	24	19.2	1.23
Hold Solenoid	16	24	19.2	1.30
Disengage Solenoid	16	24	19.4	1.20
Engage Pump	4	16	6.71	1.12
Disengage Pump	4	16	6.83	1.12
Command Monitor	364	376	370.34	2.67
Health Monitor	344	442	373.23	1.56

Table 3: Execution time of tasks in µs

.

Task	**Execution Time (c_i)**	**Period (p_i)**
Temp. Read (x8)	4224	25000
Press. Read (x2)	1424	30000
Command Monitor	376	50000
Health Monitor	442	50000

Table 4: Execution time and periods of tasks in µs

.

Schedulability Analysis

Once WCET for each task is known it's fairly straight forward to determine if the task set is schedulable within the hyper period. By calculating the LCM of the task's periods one can determine that the hyper period for this task set is 150 ms. Then using the sufficient condition for RM schedulability outlined by Liu and Layland [5]

$$U \leq n(2^{\frac{1}{n}} - 1)$$

we can verify that

$$U = \sum \frac{c_i}{p_i} = 0.208 \leq 4(2^{\frac{1}{4}} - 1) \leq .7568$$

Since the condition is satisfied, we can see that the task set is schedulable and utilizes about 21% of the available processing power. A lower bound on the system response time to commands can now be calculated by determining the minimum period for the command task such that task set is still schedulable. In order to determine this period we can use time demand analysis. In order for the task set to be RM schedulable it must satisfy the following inequalities

$$c_1 + c_2 + c_3 + c_4 \leq 25000$$
$$2c_1 + c_2 + c_3 + c_4 \leq 30000$$
$$2c_1 + 2c_2 + c_3 + c_4 \leq 50000$$

In order to determine the minimum value for the command monitor task's period we need to determine minimized p_3 that satisfies at least one of the following inequalities

$$c_1 + c_2 + c_3 \left\lceil \frac{50000}{p_3} \right\rceil + c_4 \leq 25000$$
$$2c_1 + c_2 + c_3 \left\lceil \frac{50000}{p_3} \right\rceil + c_4 \leq 30000$$
$$2c_1 + 2c_2 + c_3 \left\lceil \frac{50000}{p_3} \right\rceil + c_4 \leq 50000$$

Since only one inequality need to be satisfied we can choose the inequality with the largest right hand side and determine

$$c_3 \left\lceil \frac{50000}{p_3} \right\rceil \leq 38262$$
$$\left\lceil \frac{50000}{p_3} \right\rceil \leq 101.76$$
$$p_3 \geq 493$$

Thus, providing a lower bound on the period of the command task. This implies that theoretically the system could respond to commands within 493 µs without missing deadlines. After modifying the task set to have this new period, time demand analysis was performed on a computer to verify the result as shown in Figure 8

```
Inequality satisfied for tasks
Task     c       p
1        376         493
2        4224        25000
3        1424        30000
4        442         50000

101*c1 + 2*c2 + 2*c3 + 1*c4 <= 49793

Process finished with exit code 0
```

Figure 8: Output from time demand analysis.

Leveraging the Idle Task for Saving Power

Since the task set doesn't fully utilize the CPU, there will be a considerable amount of idle time where all tasks are either waiting or blocked. In FreeRTOS there must always exist a task that is ready to run, so at the time of scheduler initialization an idle task is always created with the lowest possible priority in the system. This idle task is then executed whenever no tasks are in the ready queue and can then be hooked into by the application developer. If the idle task is left unimplemented, it simply cycles in and out of the idle task wasting CPU cycles. An interesting use of this idle time in embedded applications is putting the system into a low power state in order to conserve energy. By using the idle task hook provided by FreeRTOS in combination with the built in sleep functionality of the ATmega2560, one can easily implement this functionality on an Arduino based system. However, the frequency of the tick interrupt can have a drastic affect on the amount of power saved using this method.

Sleep mode on AVR based processors sends the processor into a low power state until an interrupt, either external or from a timer occurs. Since the watchdog timer initiates an interrupt at every tick, during idle periods the processor is sent to sleep and woken up each time a tick interrupt occurs. If the tick period is set too low then often the power saved by entering sleep mode is offset by the amount of overhead from context switching between the scheduler and the idle task as demonstrated in Figure 9. This can of course be mitigated somewhat by lengthening the interval of the tick interrupt allowing the idle task to spend more time sleeping. However one must be careful to not set the tick interrupt too high as it could reduce system

response time. I found that a good compromise between power savings and system response time for my specific application is around 30 ms.

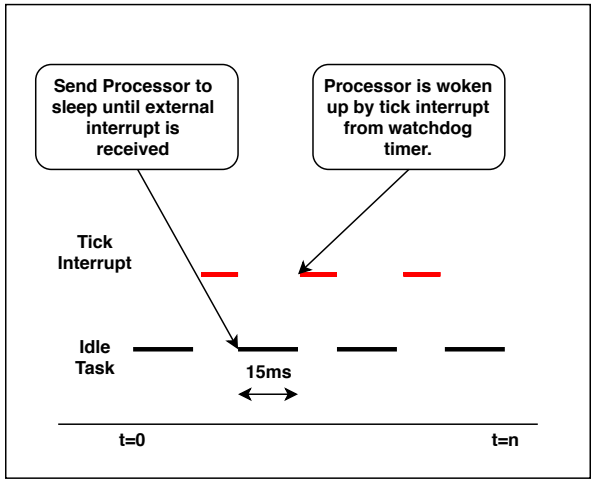

Figure 9: By default the idle task is interrupted by the tick interrupt at 15 ms intervals negating much of the power saving obtained by sending the CPU to sleep during idle periods.

AVR based processors support a variety of low power modes that can be leveraged to save power. After implementing the idle task to send the system into a low power state I then measured the maximum, minimum and average power draw over a period of one minute for each of the provided low power modes as shown in table 5. With no sleep mode defined in the idle function the system draws around 169 mA but utilizing the various sleep modes provided by the system the power draw can be reduced to a minimum of around 141 mA providing a 16.57% reduction in power draw. Note that the power draw measured here is from the entire board including the voltage regulator and on board status LEDs the actual power drawn from the CPU can be determined by subtracting the nominal current from the current draw when the CPU is shut down.

Power Saving Mode	Min	Max	Avg.	Power Saved	%
Sleep Disabled	0.168	0.170	0.169	0	0
SLEEP_MODE_IDLE	0.146	0.151	0.149	0.020	11.83
SLEEP_MODE_ADC	0.147	0.149	0.147	0.022	13.02
SLEEP_MODE_POWER_SAVE	0.148	0.151	0.149	0.020	11.83
SLEEP_MODE_POWER_STDBY	0.142	0.145	0.142	0.027	15.98
SLEEP_MODE_POWER_DOWN	0.136	0.145	0.141	0.028	16.57

Table 5: Comparison of different power modes

Power Saving

Sending the system into a low power state during the idle task is a fairly simplistic solution to saving power. Several techniques exist to solve this problem in a more elegant fashion, one of the most popular of which is Dynamic Voltage Scaling (DVS). DVS is a technique in which the clock speed and voltage of a CPU are scaled dynamically based on the slack time of a task set. Since the power function of a processor as it relates

to clock speed is convex this generally creates very good power saving results. An example of an offline low power scheduling algorithm utilizing this technique was proposed by Yao et.al [4].

While the Arduino board specifically does not support dynamic voltage scaling such a board could be developed and so it is interesting to explore how the idle shutdown method compares to a DVS technique. In the aforementioned paper an optimal scheduling algorithm was developed utilizing an intensity factor g which calculates a lower bound on average processing speed for a set of tasks on an interval $[z, z^{'}]$. The algorithm identifies a critical region that maximizes g and then schedules it on that interval before reseting the deadlines and arrival time of other tasks in that interval.

After running a simulation of this algorithm using the task set from Table 4 it was determined that the entire task set can be scheduled with a g value of 0.2096. Thus, we can run the CPU at roughly 21% of its maximum speed of 16 MHz and still meet deadlines. A feasible EDF schedule generated by the simulation is shown in Figure 10.

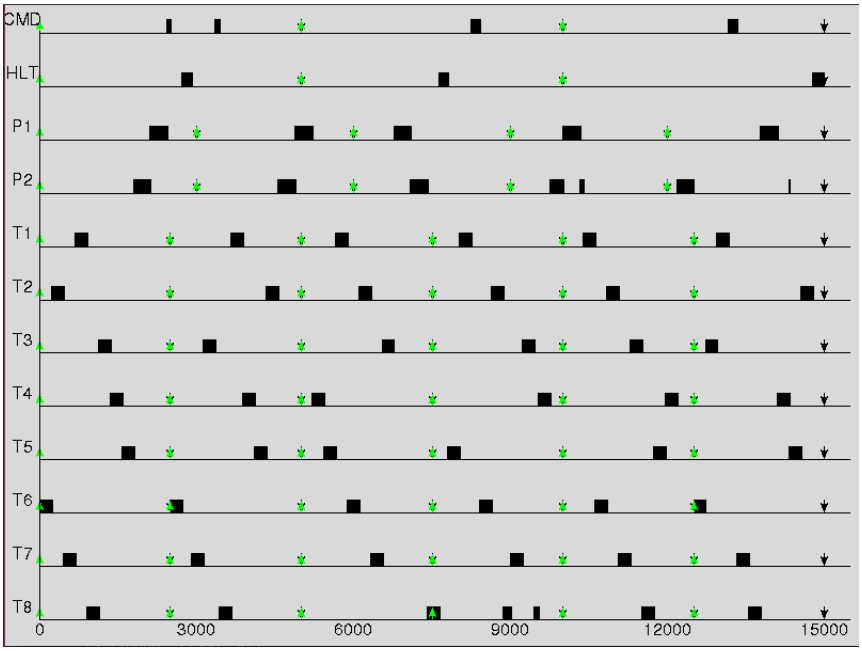

Figure 10: Feasible schedule for one hyper-period of 150 ms

Given that the task set can be feasibly scheduled at a much slower speed, it's interesting to explore what the actual performance would be at that speed. Shown in Figure 11 is the power draw from the Atmega2560 as a function of processor speed. Running at 21% of 16 MHz at 5 V shows a total power reduction of approximately 17 mA which is slightly lower than what was achieved by the idle shutdown technique. This difference is likely due to the fact that shutting down the CPU also turns off power hungry systems that are independent of the CPU clock speed such as the ADC. The graph only displays power saved specifically by the CPU and does not factor in many of the other components that could potentially be shutdown or slowed as well. This seems to imply that, for this specific application DVS would perform worst than the simple CPU idle shutdown method.

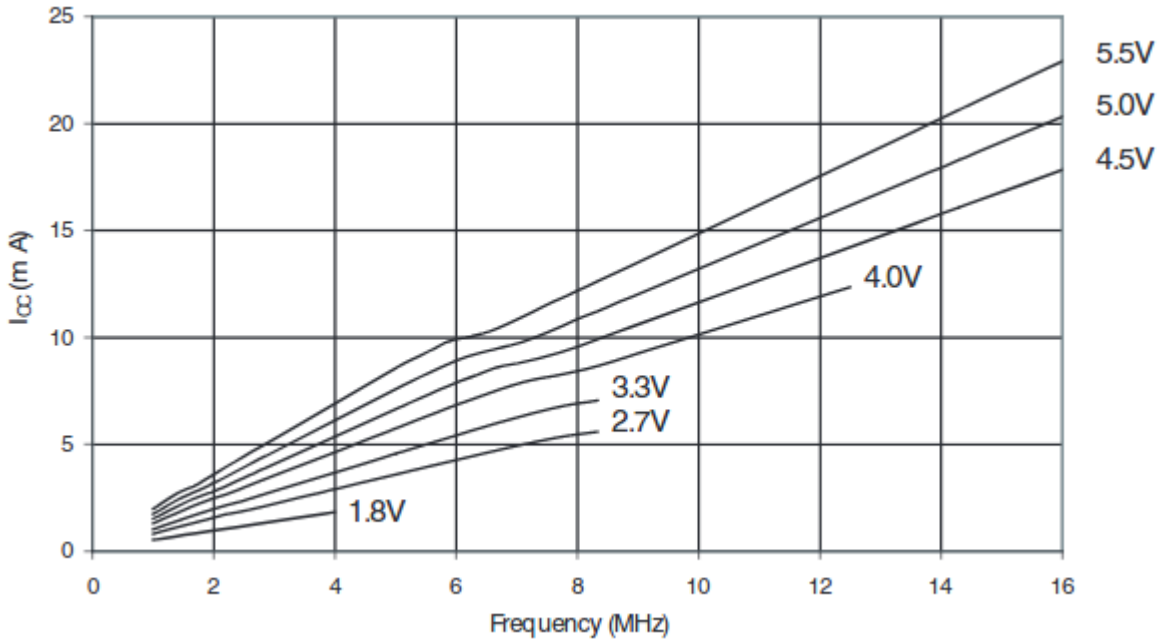

Figure 11: Active Supply Current Vs. Frequency from the Atmega2560[3] datasheet.

Conclusion

Overall, the real-time systems techniques utilized proved to be very helpful for the specification and implementation of the controls systems for the SORA payload. The system specification was proven to be correct using model checking and CTL, a theoretical lower bound on command response time of 493 µs was calculated and a 16% power saving was achieved by using a simple idle shutdown technique. Finally, a theoretical calculation on the amount of power saved by slowing the clock speed shows that more power is saved by using a CPU shutdown technique which implies that it would likely not be beneficial to switch to a board that supports DVS.

References

[1] Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1983). Automatic verification of finite state concurrent system using temporal logic specifications. Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL 83. doi:10.1145/567067.56708

[2] NuSMV Reimplementation of the SMV Modeling Language at http://nusmv.fbk.eu/

[3] Atmel Atmega 2560 datasheet: `http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf`

[4] Frances Yao, Alan Demers, Scott Shenker (1995). A Scheduling Model for Reduced CPU Energy, Xerox Palo Alto Research Center

[5] Liu, C. L.; Layland, J. (1973), "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, 20 (1): 4661, doi:10.1145/321738.321743

Appendix A: Formal Specification in SMV

```
MODULE main

VAR
    system: {executing, idle, unhealthy};
    pumps: {on, off};
    solenoids: {engaged, disengaged};
    cmd: {start, stop, none};
    error: boolean;


ASSIGN
    init(system) := idle;
    init(pumps) := off;
    init(cmd) := none;
    init(solenoids) := disengaged;

    next(system) :=
    case
cmd = start & system != unhealthy: executing;
        cmd = stop & system != unhealthy: executing;
        cmd = none: {idle, unhealthy};
        TRUE: system;
    esac;

    next(pumps) :=
    case
system = executing & cmd = start : on;
        system = executing & cmd = stop: off;
        system = unhealthy: off;
        TRUE: pumps;
    esac;

    next(solenoids) :=
    case
system = executing & cmd = start : engaged;
        system = executing & cmd = stop: disengaged;
        system = unhealthy: disengaged;
        TRUE: solenoids;
    esac;

    next(cmd) :=
    case
system = idle & next(system) != executing: {start, stop, none};
        TRUE: cmd;
    esac;
```

```
    next(error) :=
    case
system = unhealthy: TRUE;
system != unhealthy: FALSE;
    esac;

SPEC
AG(cmd = start & system != unhealthy -> AF(pumps = on & solenoids = engaged))

SPEC
AG(cmd = stop & system != unhealthy -> AF(pumps = off & solenoids = disengaged))

SPEC
AG(system = unhealthy -> AX(error));

SPEC

AG(system = unhealthy -> AF(pumps = off & solenoids = disengaged))

SPEC

AG(cmd = start -> AF(pumps = on));

SPEC

AG(cmd = stop -> AF(pumps = off));

SPEC

AG(!error | pumps = off);
```