NPS-CS-96-009

NAVAL POSTGRADUATE SCHOOL
Monterey, California

CAPS AND REAL-TIME SYSTEMS

by

George S. Whitbeck
Man-Tak Shing

September 1996

Approved for public release; distribution is unlimited.

Prepared for: Naval Postgraduate School
Monterey, CA 93943-5000

NAVAL POSTGRADUATE SCHOOL

Monterey, California

REAR ADMIRAL M. J. EVANS
Superintendent

RICHARD S. ELSTER
Provost

This report was prepared for the Naval Postgraduate School.

This report was prepared by:

George S. Whitbeck
Major, USMC

Man-Tak Shing
Associate Professor
of Computer Science

Reviewed by:

Ted Lewis
Chairman

Released by:

Dave Netzer
Dean of Research

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1996	3. REPORT TYPE AND DATES COVERED Technical Report

4. TITLE AND SUBTITLE CAPS AND REAL-TIME SYSTEMS	5. FUNDING NUMBERS n/a
6. AUTHOR(S) Whitbeck, George S. and Shing, Man-Tak	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER NPSCS-96-009
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Professor Luqi and the CAPS group	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT *(maximum 200 words)*

We recognize the need for computer science students to understand real-time systems. Many universities are only now introducing courses in this area. Frequently, the first step in mastering a concept is learning the jargon of that particular community of professionals; the real-time community is no exception.

The Naval Postgraduate School's computer-aided prototyping system (CAPS) is an integrated set of software tools made for the rapid construction of real-time software prototypes. Students who learn this system must first have a basic understanding of the real-time world. This paper introduces students to that world and then ties it in with CAPS.

14. SUBJECT TERMS CAPS, computer-aided prototyping system, real-time systems, teaching, students	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

CAPS AND REAL-TIME SYSTEMS

Major George S. Whitbeck, USMC
whitbeck@cs.nps.navy.mil

and

Dr. Man-tak Shing
Code CS/SH
Naval Postgraduate School
Monterey, CA 93943
mantak@cs.nps.navy.mil

ABSTRACT

We recognize the need for computer science students to understand real-time systems. Many universities are only now introducing courses in this area. Frequently, the first step in mastering a concept is learning the jargon of that particular community of professionals; the real-time community is no exception.

The Naval Postgraduate School's computer-aided prototyping system (CAPS) is an integrated set of software tools made for the rapid construction of real-time software prototypes. Students who learn this system must first have a basic understanding of the real-time world. This paper introduces students to that world and then ties it in with CAPS.

CAPS AND REAL-TIME SYSTEMS

A. INTRODUCTION

This paper is designed as a primer to the Computer-Aided Prototyping System (CAPS) student who is ready to dive into the real-time aspects of a CAPS prototype. We start with a discussion of classical real-time systems and then move to how the CAPS environment creates a real-time prototype. No previous knowledge of real-time systems is needed but we do assume the reader has a some exposure to CAPS.

B. REAL-TIME SYSTEMS DEFINED

There are as many definitions of real-time systems as there are authors of real-time articles. Stankovic's definition is as close to the norm as we've seen:

> "In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced" (St88).

If these timing constraints are not met, then you have a failure. Hence, it is essential that the timing constraints of the system are guaranteed. Guaranteeing timing behavior requires that the system be predictable and reliable. Gillies (Gi95) gives an example of a robot that has to pick up something from a conveyor belt. The piece is moving, and the robot has a small window in which to pick up the object. If the robot is late, the piece won't be there anymore, and thus the job will have been done incorrectly, even though the robot went to the right place. In a military system, imagine a fighter jet with an enemy aircraft crossing the pilot's cross hairs for a fleeting moment. The system designers specified a tolerance for how fast the gun should fire once pressure has been applied to the trigger. If the plane ran on a distributed operating system that was too busy sampling airspeed to handle the trigger squeeze in a timely manner, then the fleeting target

would be gone and we'd declare system failure even though the gun fired just a few seconds late. Needless to say, testing these reactions to input and measuring how well the system responds is a critical part of the testing phase.

Now you may be thinking that all computer systems must respond in some reasonable amount of time and that even if the system doesn't meet a time constraint that the consequences won't always be disastrous. You're right and to handle those not so time critical missions, real-time systems are further categorized as follows:

> A system where "performance is *degraded but not destroyed* by failure to meet a response time constraint is referred to as a soft real-time system." Systems where "failure to meet a response time constraint leads to *system failure* is a hard real-time system" (La93).

Although we'll stick with the categories above, the definitions of "hard" and "soft" are not without controversy. Some computing professionals differentiate "hard" and "soft" by the *degree of time constraints*. For instance, a real-time process attempting to recognize images may have only a few hundred microseconds to resolve each image, but a process that attempts to position a servo-motor may have tens of milliseconds to process its data (Gi95).

Lest you think that hard real-time systems are "perfect" or "bug free" systems, understand that that's not the meaning albeit a goal. Although real-time systems are said to run precisely within bounds, almost any nontrivial software program that takes external inputs can be overwhelmed. Therefore, the goal is that the software will *operate without failure for a specified period of time*. Laplante (La93) gives a example where NASA has suggested that computers used in civilian fly-by-wire aircraft have a failure probability of no more than 10^{-9} per hour.

David Ripps explains that "a fundamental property of real-time systems is that some or all of its input arrives from the outside world asynchronously with respect to any

work that the program is already doing" (Ri89). Our "Murphy's Law" version of this is that interrupts will hit the system randomly or at the most inopportune time, and the system must handle them in stride. The program must be able to block its current activity and then execute some other task such as capturing a fleeting signal. When done, it must return gracefully to what it was previously doing . Executing several tasks in what appears to be in parallel is called "multitasking" and is a key characteristic of all real-time systems. We say "appears to be in parallel" because parallelism depends on the number of CPUs working and to which processes they've been assigned.

C. COMMON MISCONCEPTIONS

Now that we've established what a real-time system is, let's look at misconceptions. Even among computer professionals, it's common to have misconceptions about real-time systems. For instance, some have speculated that *advances in supercomputer hardware* will take care of real-time requirements. According to (St88), this is the same flawed argument that has been around for years. The history of computing shows that as machines get faster and memory gets bigger, software fills the void and, in fact, gets hungrier eventually requiring faster CPUs and more memory.

Others have said that these problems are *no different* from other aspects of computer science. However, problems in analyzing real-time systems are unique. For example, typical performance engineering in software has been concerned mostly with analyzing the average values of performance parameters. An example here is when a data base marketing team tells the design team that customers lose their patience if they have to wait more that a three seconds for a "save" function. The designers and testers will put the application through several strenuous tests that ensures that *on the average* a save function takes under 3 seconds. Whereas, according to (St88), an important consideration in real-time system design is whether or not some stringent deadlines can be met *in all cases*, not just in the average case .

One will occasionally see references to "real-time" systems when what is meant is "*on-line*", or "an interactive system with better response time than we used to have." Often, this is just marketing hype. For instance, although some have queried whether running "rn" (read news) is real-time, it is not, as it is interacting with a human who can tolerate hundreds of milliseconds of delays without a problem (Gi95). The bottom line is that information retrieval systems are usually not real-time.

One may also see references to real-time systems when what is meant is just *fast*. Remember, that "real-time" is not necessarily synonymous with "fast". It isn't the latency of the response per se that is at issue (it could be on the order of seconds), but "the fact that a bounded latency sufficient to solve the problem at hand is guaranteed by the system. In particular, it's frequently so that algorithms that guarantee bounded latency responses are less efficient," and thus slower, than algorithms that don't (Gi95).

D. PARTS OF A REAL-TIME SYSTEM

To take apart a real-time system, the most obvious dissection is into hardware and software. Although real-time hardware is beyond the scope of this discussion, understand that many military real-time systems designed from the bottom up use special hardware to assist in making them real-time. Additionally, add on hardware can be used to test timing constraints as discussed later.

Until now, when we've said "real-time system," we've included all the software involved. At this point, let's make it clear that real-time software generally comes in two flavors. The first is the complete real-time software written from scratch where there is little to no distinction between the application and the operating system. A control system on a one-of-a-kind satellite is one example. These, of course, are expensive and generally don't follow the principle of software reuse. The other is one where a real-time application is written for an established operating system; in practice, that operating system would be a real-time operating system (RTOS). At this point, let's further

examine a RTOS.

The economics and reliability of a reusable operating system offer a tremendous incentive into the research and development of robust and flexible RTOSs. Application designers can then choose one of these off-the-shelf RTOSs upon which they can add their real-time application. Then, the RTOS "is tuned and sculpted by the application, much as an athlete is trained for a specific event" (St92). The idea of an application "tuning" an operating system is unusual unless one is discussing real-time applications running on a RTOS. In fact, most operating systems protect themselves from the users but the RTOS specifically allows itself to be *user controlled* to a degree. Kevin Morgan, writing in (Mo92), says that this idea of user control, along with the following four other general attribute areas, are what distinguish a RTOS from a normal operating system. His second characteristic, *determinism,* is "the tendency of a system to perform an operation in a well-defined, or "determined" time period." An operating system that is fully deterministic will perform every operation in the same amount of time regardless of what else is going on in the system. No operating system is fully deterministic but RTOSs are much more deterministic than conventional ones. *Responsiveness* is the third area. This is the ability of an operating system to respond quickly to an event such as an interrupt. A fighter pilot has only moments to react to the warning of an incoming missile. If the sensor that picked up the missile can't interrupt the operating system in a timely manner (as set forth in the system specifications), then the pilot will have less time to react and the results could be disastrous. A fourth area mentioned by Morgan is *reliability* or what we call crash resistance. The standard here must be far stronger than for a normal operating system, and ideally, a RTOS is able to preallocate required resources. The final distinguishing characteristic is *fail-soft* operation. When UNIX detects a corruption of data in the kernel, it performs a panic operations and shuts down fast. If a RTOS sees a crash coming (and it should), it must gracefully degrade - sort of a soft landing - and perform a predictable behavior like sounding an alarm, starting up a backup system, or better yet, just notify the specific process of the problem and then operate in a degraded mode (Mo92).

A RTOS that appears to meet these criteria is "Real-Time UNIX." There are several versions of it with specific enhancements for real-time operations, most notably, kernel preemption. Philip Laplante explains the following:

> "In standard UNIX, a process which makes system calls is not preemptable. Even if the calling process is of low priority, it continues execution until it is stopped or completed. In real-time UNIX, preemption points have been built into the kernel so that system calls can be preempted without running to completion. This radically reduces response times (La93).

Moving from a RTOS to a *distributed RTOS* is a significant step forward. Stankovic says that much "research is currently being done on developing time-constrained communication protocols to serve as a platform for supporting user-level end-to-end timing requirements" (St92). He and others have active research projects such as the Mars approach and the Spring kernel. Concerning the structure of the network needed, Zhoa and colleagues writing in (Rt94) suggest that the fiber distributed data interface (FDDI) is "a fine candidate for mission critical real-time applications, due not only to its high bandwidth, but also to its property of bounded token rotation time and its dual ring architecture." The bottom line today, however, is that since a virtual uniprocessor is still in research, it seems accurate to say that a true distributed RTOS is a ways off.

What the reader should take away concerning a distributed real-time system is that there is a change in who's driving the requirements train. In a single computer real-time system, there will usually be studies from subject matter experts that tell the designers what timing constraints are on each operation. For instance, there may be a study that says that if a fighter pilot hopes to hit a crossing enemy aircraft with his 20mm guns, then the firing pin on the gun must strike the primer no later than 200 ms after the pilot pulls

the trigger. This is fairly straightforward and can easily be put into the requirements document.

However, if that same airplane is run on a distributed operating system, like a token ring, then the designers must examine that system and ensure the both the trigger's and gun's computers will get the token quickly enough to meet the 200ms deadline. This can, of course, be engineered into the problem but the complexity of the systems is obviously increased.

E. HOW CAPS BUILDS A REAL-TIME SYSTEM

1. The Real-time Ship Captain

To build our knowledge of how CAPS schedules time critical tasks, we'll create an imaginary ship's Combat Information Center (CIC). The Captain sits in the middle of this room and has sailors at four consoles to his front: damage control, air, surface, and subsurface. He's very concerned about these operations so he rotates his time through each of them once every 12 minutes. However, he also has people who need to occasionally give him messages, so those people can press a doorbell button from outside the CIC that turns on a red light on the captain's armrest. From the moment the light comes on, the captain has guaranteed that he will let them in and read the message within 8 minutes. His plan is to check the light at least every 6 minutes and then dedicate no more than 2 minutes to reading the message. Given the above requirements, he's developed the following routine:

- get a damage control report,
- then an air report,
- then he checks the red light, and, if it is on, he gets and reads a message,
- then a surface report,
- then a subsurface report, and finally,
- he again checks the red light and reads a waiting message, if one exists.

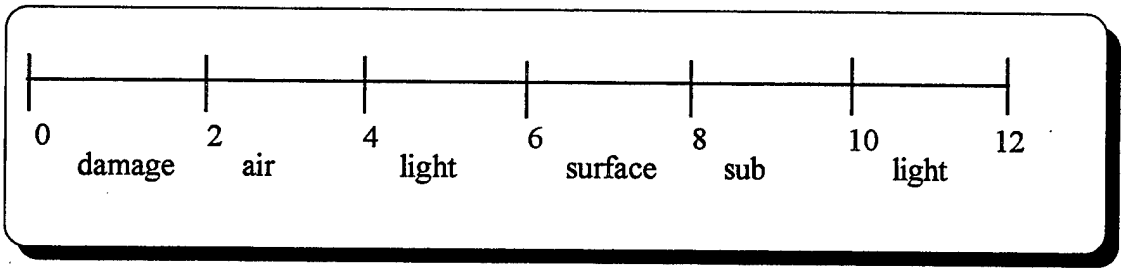

Figure 1

Once complete, he repeats the cycle. The length of this cycle is the *schedule length*. (This scenario uses CAPS terms (*italicized*) as found in (Lu92) and (Lu96).)

As shown in figure 1, when the captain gets a console report from a sailor, he wants to finish within 2 minutes. That amount of time is the *finish within* for each of the console operations. However, for our requirements he doesn't want the sailor to talk for the entire 2 minutes; the captain needs time to check the console himself and to ask questions. Therefore, he gives each sailor no more than 90 seconds for a formal report during the 2 minutes. The 90 seconds is the sailor's *maximum execution time* (MET).

Note that the sailor may not be ready to speak the moment he's called on. The sailor will satisfy the captain's schedule as long as he begins the 90 second maximum length report no later than 30 seconds after the sailor is called on. For simplicity, these numbers are the same for each operator in our scenario. Additionally, we don't address what the captain does with his spare time if, for instance, the report is shorter than 90 seconds. But suffice it to say, he uses free time to do non-time critical tasks.

His attention to each of the console operators is referred to as a *periodic operation*. The name comes from the fact that they are triggered by a time event. However, when we shift the discussion from the console operators and begin to talk about the captain's armrest light, then the names change. In this new paradigm, the tasks are called *sporadic operations* and they are operations triggered by an external event (an

event external to the current Ada procedure). The chief concern here is ensuring the captain gets to the interruption fast enough to read the report in the time he's allotted. The worst case condition occurs if the captain checks the light and it's off; it will then be just under 6 minutes before he checks it again and then no more than 2 minutes to read the message. We calculate this as follows:

1. Check the light --near instantaneous event -- and suppose it's off.
2. But, just after he turns his head away from the light, it comes on.
3. Now he does some thinking for the remainder of the 2 minutes allotted to handling the interruption.
4. He then spend 4 minutes getting briefed on two of the consoles.
5. Next, he checks the light again and this time it's on.
6. Finally, he lets the messenger in and reads the message.

This (almost) 8 minutes is the *maximum response time* (MRT) and is the upper limit (worst case) on the time between the light coming on and the captain completing his reading of the message. The MRT is like of a window of action to notice the light and read the message. The captain's reading of the message (no more than 2 minutes in our scenario) is his MET and is the action portion of the window. A new term, the *minimum calling period* (MCP), is the shortest time allowed between two successive pressings of the doorbell. If unspecified, the MCP equals MRT-MET (i.e., 6 = 8 - 2). (Lu96) We will explain its usage later.

2. CAPS Tasks

Atomic operators in a CAPS data flow diagram become Ada procedures in the CAPS implementation. Prioritizing these operations into time critical (high priority) and non-time critical (low priority) is fundamental to a real-time prototype specification. So how does one know from looking at a CAPS data flow diagram if an operator is time critical or non-time critical? Figure 2 below shows a segment of the augmented data flow diagram that models the CIC scenario. Note that operator "READ_DAMAGE_REPORT" has a MET of 90 seconds assigned to it. The MET is the longest time between the instant

the operator begins execution and instant it completes execution. (Lu92) The presence of this MET means that the operator will be treated as a time critical operation. The absence of a MET, such as in operator "MAKE_NOTES" below, means that CAPS will treat the operator as a non-time critical operation. (Lu96)

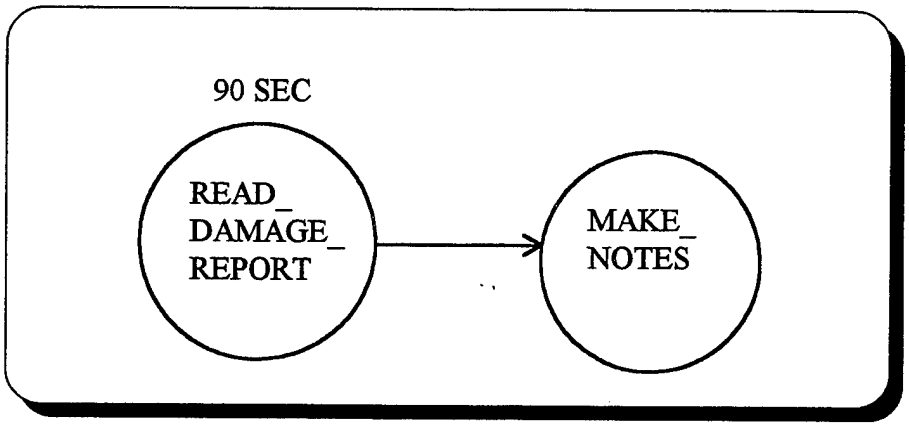

Figure 2

3. Time Constraint Options

Notice in the Venn diagram below, Figure 3, that the domain of tasks are broken down into the two categories: Time Critical and Non-Time Critical. Expanding upon that further, notice also that there are asterisks on three of the labels. These are the only three *time constraint* options the CAPS user has for operators: *Periodic*, *Sporadic*, and *Non-time Critical*.

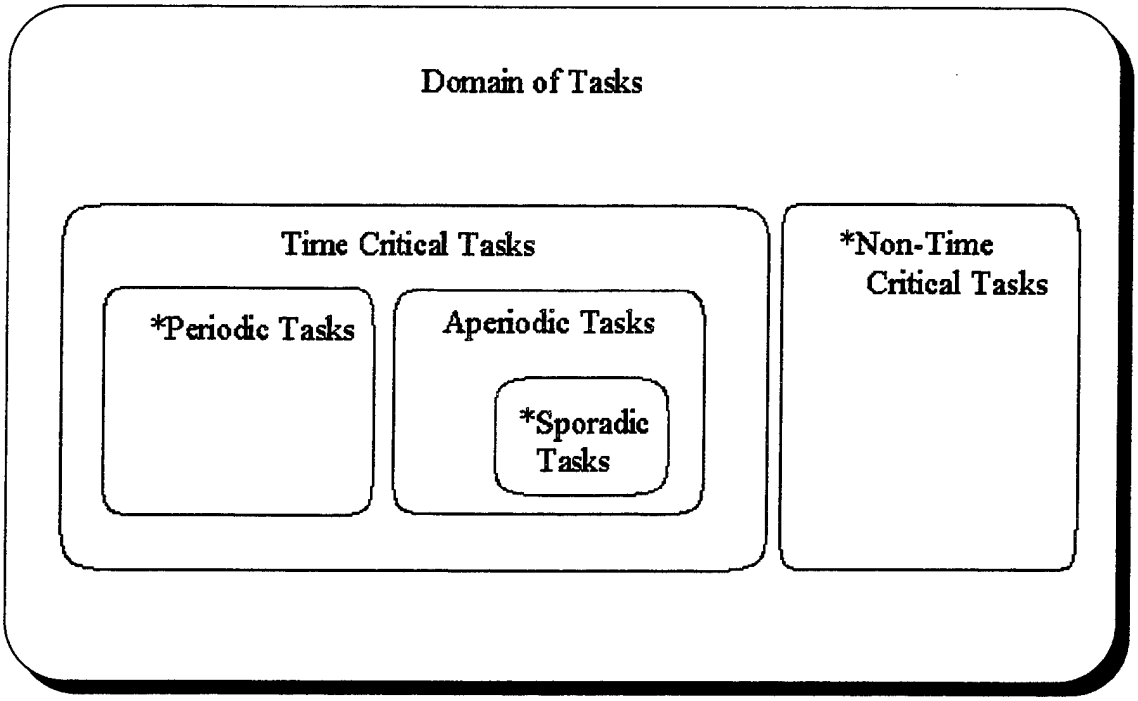

Figure 3

a. Periodic and Sporadic Tasks

A time critical task (periodic or sporadic) is one that has a timing constraint associated with it. In CAPS, an operator having a MET is considered time critical and will be scheduled in the static schedule loop of the "<prototype>.a" file. These tasks are considered HIGH priority.

Now let's address triggering. There are only two ways to trigger time critical tasks. The first is using a *time event* and these tasks are called *periodic*. An example is an airborne phased array radar like on the JSTARS aircraft that scans the battlefield once every two seconds. (Two seconds is just an example; the actual number is classified.) The second way to trigger a task is with a *physical event* and these tasks are called *aperiodic*. An example is using a mouse to press a Quit button on a GUI. A subset of aperiodic tasks are *sporadic* tasks. The distinction is subtle and will be discussed shortly.

Periodic tasks are fairly intuitive. The requirements analysis will normally drive the size of the period. In a hard real-time system, failure to meet this timing constraint will, by definition, lead to system failure. The CAPS scheduler starts with periodic tasks when it builds the prototype's static schedule.

Aperiodic tasks are essentially random tasks triggered by some external event such as pressing a mouse button or detecting a hardware interrupt. (Co95) The program just looks for that event every so often, say every half second, and then executes the associated task in the required MET if such an event is found. This rate is the "trigger period" of the aperiodic event. Aperiodic task are not run at a set period but they are looked for at a set period. If found, they are then executed. However, it should also be clear that where a normal computer may just ignore an event if it happened over and over again at a rate faster than the computer could keep up with, a hard real-time system should be built to handle a defined worst case situation. So how does CAPS schedule this worst case situation of repetitive inputs without overflow? It does it with a special kind of aperiodic task called a *sporadic task*. Sporadic tasks are aperiodic tasks in which a minimum calling period between any two aperiodic events is required. (Co95) In CAPS, sporadic tasks are triggered by the arrival of data on data streams. Put another way, the only method a hard real-time system has of guaranteeing no overflow for aperiodic events is by restricting the rate in which the events are generated. (The minimum calling period is usually derived from the application domain and it indicates the maximum load the real-time system has to handle.) In our ship's captain example, the captain will impose a MCP of at least 6 minutes in order to guarantee no external event overflow based on his schedule in figure 1. So in other words, the system could break down if he gets two external messages within 6 minutes of each other. However, that shouldn't happen if he evaluated his requirements properly.

b. Non-time Critical Tasks

A non-time critical task is one that does not have a timing constraint associated

with it. In CAPS, an operator without a MET is considered non-time critical and will be scheduled in the dynamic schedule loop of the "<prototype>.a" file. These tasks are considered LOW priority and are run in the gaps in the static schedule. In essence, the static schedule loop turns over control to the dynamic schedule loop when there are no high priority tasks to run. The dynamic schedule loop now runs any and all non-time critical tasks if and when their triggering conditions are satisfied. The static schedule loop will preempt the dynamic schedule loop whenever a high priority task is ready to run.

F. HOW CAPS PROTOTYPES RUN IN REAL-TIME

1. CAPS Prototypes in the Real-time Spectrum

It's important to note the spectrum of real-time systems. From most commercially specialized to least, we propose this as a simple ordering:

1. embedded systems -- real-time code embedded onto chip
2. The real-time application and operating system functionality coded together as a system running on a workstation
3. real-time program running on top of a RTOS on a workstation
4. real-time prototype running on top of a RTOS on a workstation
5. real-time prototype running on top of UNIX on a workstation

The differences between numbers (1) through (3) should be understandable after having read paragraphs A through F. However, the distinction between the remaining systems needs explanation. The separation between a real-time program (3) and a real-time prototype (4) is meaningful only for "throw away" prototypes. In such cases, the prototype constructed in (4) is solely for the purpose of firming up requirements. The real-time system has to go through another phase of software architectural design and implementation to evolve to the production quality program of (3).

As an academic research project, CAPS produces prototypes at (5). This allows tremendous portability to networks around the country for teaching purposes. Computer

Science graduate students in most academic institutions work on and are familiar with UNIX workstations. They are pleased since they don't need to learn the intricacies on a new operating system; they can focus their attention on learning CAPS.

2. Moving Up from (5) to (4)

Another issue is "What makes a CAPS prototype run in real-time if it is run on UNIX instead of a RTOS?" The answer is the Ada Runtime Executive. It was designed with real-time in mind and controls the static and dynamic schedules. Although we've made a case for having CAPS prototypes run on normal UNIX, there is at least one strong incentive for moving up to running on a RTOS. And that is to accurately test timing constraints. While the CAPS prototype tries to accurately measure timing and report violations, the approach in our view is less than optimal if run on a normal UNIX operating system. As just mentioned, the Ada runtime executive controls all the real-time tasks in a CAPS prototype, and it also measures time usage and reports timing violations. But, if UNIX blocks the prototype to let some other process run, the Ada runtime executive never knows that it was put to sleep. When it compares start and stop time stamps for an operator (Ada procedure), it assumes there was no unforeseen break and the prototype thus produces a timing error message. Hence, on a day when the workstation is under a heavy load, the prototype may have several timing violations, but on a light day, it may run without any. This is important only if the designer is looking to CAPS to validate timing constraints. When running a prototype, a lack of timing errors implies that the constraints are valid. However, the presence of timing errors does not necessarily mean that the programmed timing constraints are in error. It could be that UNIX preempted the prototype as mentioned above.

A simple solution would be to run the prototype on Real-time UNIX or some other RTOS as mentioned previously (i.e., move up our list from 5 to 4). This solution is simple in theory but not necessarily easy to implement. The first and most formidable challenge is to find a RTOS with a matching Ada compiler and ensuring that both are

compatible with your workstations. We recommend this as future thesis work for the CAPS group at the Naval Postgraduate School.

G. CONCLUSION

Like software engineering in general, real-time system engineering is still in its infancy. Many universities are only now introducing it into their curriculum. Likewise, government and industry training departments are recognizing the need for professionals versed in real-time systems. As these schools and departments look for low cost and innovative ways to teach this challenging subject, they'll want a hands on tool to augment classroom lectures. Their students must be able to apply at their workstations what they've learned in the classroom if they hope to understand the real-time domain. CAPS is an excellent tool for both teaching and prototyping hard real-time systems.

ACKNOWLEDGMENTS.

This research was supported in part by the National Science Foundation under grant number CCR-9058543, the Army Research Office under grant number ARO 111-95, and by the Naval Postgraduate School via the Direct Funding Research Program.

Additionally, we would like to thanks Dr. Luqi of the Naval Postgraduate School for her tremendous leadership in the field of real-time systems.

End notes

(Co95) M. Cordeiro, Distributed Hard Real-time Scheduling for a Software Prototyping Environment, Ph.D. Dissertation, Naval Postgraduate School, March 1995.

(Gi95) Gillies, D. *FAQ page for the Real-Time usenet group,* http://www.cis.ohio-state.edu/hypertext/faq/usenet/realtime-computing/faq/faq-doc-4.html

(La93) Laplante, P. A. *Real-Time Systems Design and Analysis.* Piscataway, New Jersey: IEEE Press, 1993

(Lu92), "Computer-Aided Prototyping for a Command-and-Control System Using CAPS", *IEEE Software*, Jan 92.

(Lu96) Luqi, Class Notes for CS4920 -- Rapid Prototyping, Naval Postgraduate School, April 1996.

(Mo92) Morgan, K. D. *The RTOS Difference*. Byte, Aug. 92.

(Rt94) Rtoss. *Real-Time Operating Systems and Software*. Los Alamitos, California: IEEE Computer Society Press, 1994.

(St88) Stankovic, J. A. *Misconceptions about Real-Time Computing. A Serious Problem for Next-Generation Systems*. Computer, Oct. 88.

(St92) Stankovic, J. A. *Real-Time Computing.* Byte, Aug 92.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   8725 John J Kingman Road, Suite 0944
   Fort Belvoir, Virginia 22060

2. Dudley Knox Library — 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, California 93943

3. Director, Training and Education — 1
   MCCDC, Code C46
   1019 Elliot Rd
   Quantico, Virginia 22134-5027

4. Dr. Luqi, Code CS/Lq — 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5002

5. Dr. Shing, Code CS/Sh — 50
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5002

6. Currilicular Officer, Code 32 — 1
   Naval Postgraduate School
   Monterey, California 93943-5002

7. Margaret Gates — 1
   2266 Sequoia Drive
   Clearwater, Florida 34623

8. Maj. George Whitbeck — 2
   107 Coopers Ln
   Stafford, Virginia 22554