dowelldesign

Real-Time Across the Organization

A Look at Real-Time Computing Systems and the Impact of Using Real-Time on Various Roles in the Project Development Team.

Introduction

Generally speaking, real-time systems are those that must react within certain time constraints to events in their environments. This means that the correct behavior of such systems will depend not only on the result of a computation but also on the time at which the result is produced.

The term real-time has its origins in the context of simulation. A simulation is a process that attempts to mimic certain key characteristics of an underlying system, called the target system. One of the key dimensions of the problem is the relationship between the progression of time in the simulation and the progression of time in the real-world as experienced by the people running the simulation. There are three possibilities: time can run faster in the simulation than it does in the "real" world, it can run at the same rate, or it can run slower. Consider the following illustrations of the three cases:

- A weather simulation: if the goal of a simulation is to predict the path of a hurricane over a week, the simulation is not going to be very useful if it takes two weeks to complete
- A flight simulation: if the goal is to provide a realistic model of an aircraft in its environment, it is important that events in the simulation proceed at the same rate as they would in the real aircraft.
- A physics simulation: if the goal is to simulate a physical phenomenon, it may be unnecessary or infeasible to execute the simulation anywhere near real-time. The simulation must finish in a reasonable amount of time to be useful, however.

All three of these cases impose constraints on the progression of simulated and target system times; but not all three would be considered real-time systems.

The weather and physics simulation cases illustrate what is known as to High Performance Computing (HPC). The more computing power that can be applied to the problem, the more detailed and accurate simulations will be; and the earlier the results will be available. The benefits of such simulations slowly decrease as the simulation takes longer and longer to execute. In the case of the weather simulation, there is an instant where the result can no longer be called a prediction, but the result will not be considered very useful even at that point. In the case of the physics simulation, any timeliness constraint is really based on the user's patience level.

The main characteristic that distinguishes real-time computing from other types is the relative importance of time. The flight simulation example places a great deal of importance on the relationship between simulation time and real-time. If the simulation does not accurately reflect how a target aircraft behaves, it is considered incorrect – it is incorrect if it runs either faster or slower than the external, or real-time. If the weather simulation provides accurate output and executes faster than real time, it is considered useful. It may be considered more useful if it is faster. The physics simulation is considered useful if it can finish before the patience of the user is exhausted.

Notice that speed or performance is not considered a necessarily important criterion in determining what is and what is not a real-time system. In many fields, the concepts of real-time and performance are confused. A system is not necessarily a real-time system if it is "fast," and conversely a system is not automatically rejected from being real-time if it is relatively "slow." The requirements of a control system are derived from its

environment and specification; and not by some artificial property called speed. It turns out that it is much more important that a real-time system be predictable than arbitrarily fast.

The key difference between a real-time system and a non-real-time system is the presence of a deadline.

> Definition 1: A *deadline* is a maximum time within which a system task must complete its execution.

In critical applications, it is possible that a task which produces a result after a deadline is not just late, it is wrong!

> Definition 2: *Real-time systems* are those in which overall correctness depends on both functional correctness and timing correctness.

Depending on the consequences that may happen if a result is produced late, real-time systems are usually separated into two classes:

> Definition 3: A real-time deadline is said to be *hard* if missing that deadline may cause catastrophic consequences in the environment under control.

> Definition 4: A real-time deadline is said to be *soft* if meeting the deadline is desirable but does not cause catastrophic damage and does not jeopardize correct behavior.

Performance enters the picture when a given amount of processing must be done in order to meet a deadline. For example, if the hypothetical flight simulator can correctly fulfill its requirements with a given amount of computing power, adding more computing power does not make it more real-time; and reducing computing power does not somehow make it less real-time. If, however, the system cannot keep up with required processing it will not be a feasible real-time system unless enough computing power is applied to perform the required computation in the desired time.

In the remainder of this document, two case studies will be made to illustrate both the physical scale of existing real-time systems and also to illustrate the concepts of hard and soft real-time systems, and how they may be realized.

The Apple Computer iPod serves as an example of a small embedded system with soft real-time constraints. The Large Hadron Collider at the European Organization for Nuclear Research (CERN) illustrates that real-time systems can be quite large and satisfy hard real-time constraints as well. We also show that the operating system technology behind both of these systems is actually within reach of anyone with access to the Internet.

The Apple iPod

It is probably safe to assume that most people believe the idea of a portable audio player originated in a moment of brilliance by Apple CEO Steve Jobs; and the resulting iPod was designed entirely by Apple engineers in California. When Apple introduced the iPod in 2001, Jobs said, “with iPod, Apple has invented a whole new category of digital music player that lets you put your entire music collection in your pocket and listen to it wherever you go."

Despite the popular conception, the idea of a portable media player has been around since 1979. Figure 1 shows the original sketch of a media player concept patented by Kane Kramer in 1979. As in so many stories, the original patent for this device expired in 1999 and Kane Kramer did not have the financial resources to renew it. Kane never profited from his idea, and the patent expiration freed the market and led to an explosion of MP3 players appearing around 1999.

The kernel of the idea that became the iPod and the iTunes music store originated with an independent contractor named Tony Fadell. "Tony's idea was to take an MP3 player, build a Napster music sale service to complement it, and build a company around it" [5]. Fadell pitched his idea to a number of companies, but the only one who listened was Apple. Apple found existing digital music players, "big and clunky or small and useless" [4]. Sensing an opportunity, Apple hired Fadell in early 2001 and formed a design of roughly 30 people, including designers, programmers and hardware engineers. This team produced the first iPod in less than a year.

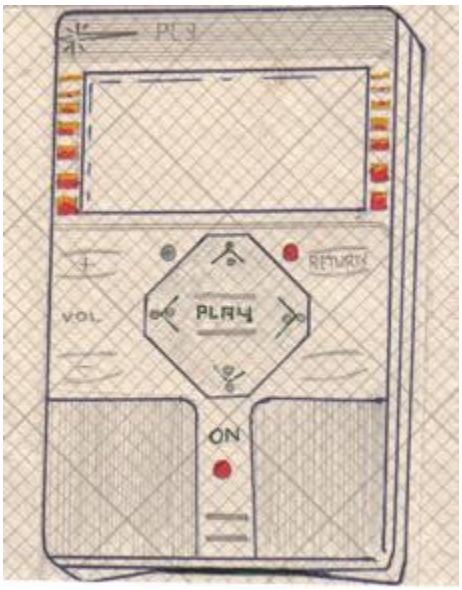

Figure 1. A sketch of what is claimed to be the first conceptual design for a digital audio player from 1979. Note the overall form factor and the central "D-Pad" control. Taken from "Kane Kramer" <http://www.kanekramer.com/html/development.htm>

As is typical at Apple, the product design was driven from the outside-in. The company had an idea of what the product should be and what it should look like, perhaps derived from the Kramer design. The initial concept along with availability of suitable components quickly locked in some of the major components including the battery and disk drive. The remainder of the hardware design was based on a reference platform provided by PortalPlayer, of Santa Clara, CA, [3]. It may be surprising to many people, but Apple provided the industrial design and specified the user interface for the original iPod; but otherwise acted primarily as a system integrator. Figure 2 shows the original iPod as it was introduced in October of 2001.

PortalPlayer was attractive to Apple because they had the software and the hardware already done, and Apple was on a tight schedule. [5]. PortalPlayer, in turn, had selected Wolfson Microelectronics to provide the codec, and Quadros Systems' Real Time eXecutive written in C (RTXC) [6] as the underlying Real-Time Operating System. The hardware used was the PortalPlayer PP5002 system-on-a-chip reference platform [7].

The PP5002 is based on an Application Specific Integrated Circuit (ASIC) featuring dual ARM7TDMI processors, USB , LCD and I^2C communication interfaces, integrated flash memory controller, direct glue-logic free interfaces to hard disk drives and support for both AC97 and I^2C codecs. Apple also contracted with a small company, Pixo, to provide a user interface and other software such as synchronization (download) code.

Figure 2. An image of a first-generation Apple iPod taken from Wikipedia <http://en.wikipedia .org/wiki/IPod>

Rather than being an exotic hardware and software environment, students in real-time or embedded systems classes will be quite familiar with similar reference platforms, such as the freely available MicroC/OS II embedded RTOS running on NXP ARM7TDMI processors in inexpensive Olimex reference boards. Although the internals of the iPod are closely held by Apple, it is interesting to compare what is known of the system to other well-known systems.

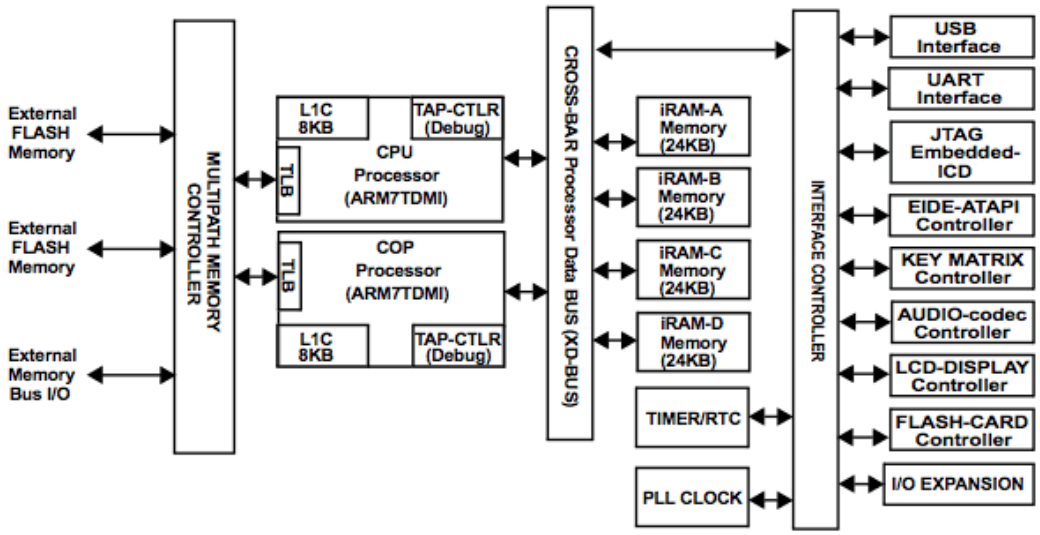

Figure 3: Block diagram of the PortalPlayer PP5002 System on a Chip ASIC as used in the Apple iPod.

The block diagram of the PortalPlayer PP5002 used in the iPod is shown in Figure 3, and the Philips NXP LPC2378 used in the Olimex reference board is shown in Figure 4.

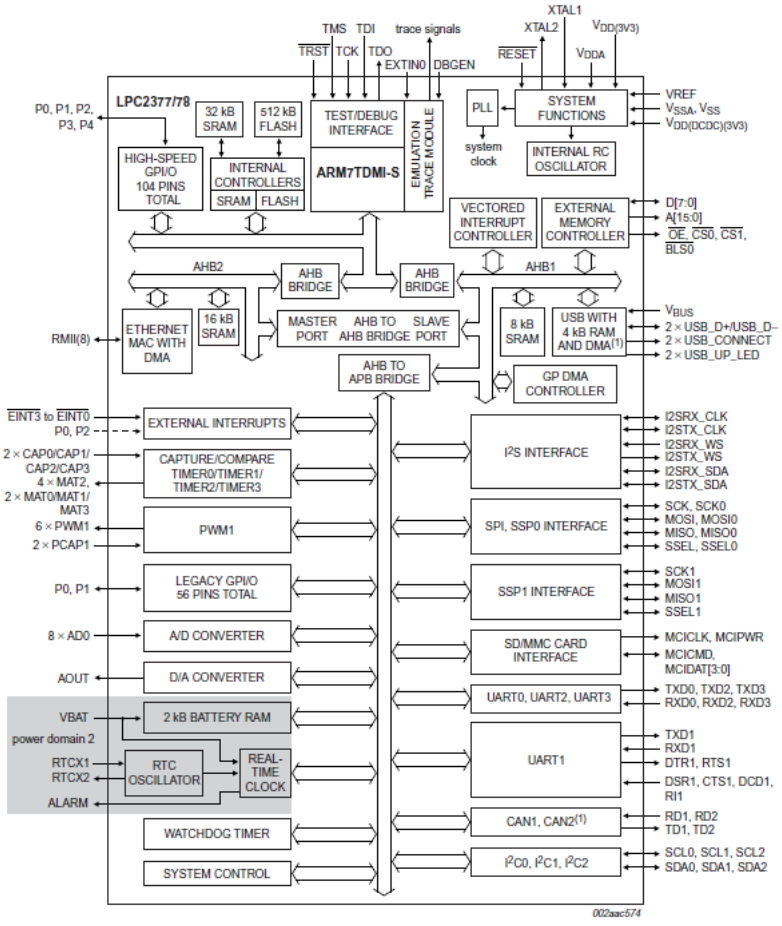

Figure 4: Block diagram of the NXP LPC2378 ASIC as used in the Olimex Reference Board

It is still within the reach of a hobbyist to prototype a product using essentially the same tools as were used to develop the original Apple iPod. Although PortalPlayer has been acquired by NVIDIA, products essentially equivalent to the original iPod reference platform are available. The operating system used in the Apple iPod is the Quadros Systems' Real Time eXecutive written in C (RTXC). Quadros curently has a close relationship with Embedded Artists who provide development kits for quick prototyping. Figure 5 shows the LPC2478 Developers kit from Embedded Artists.

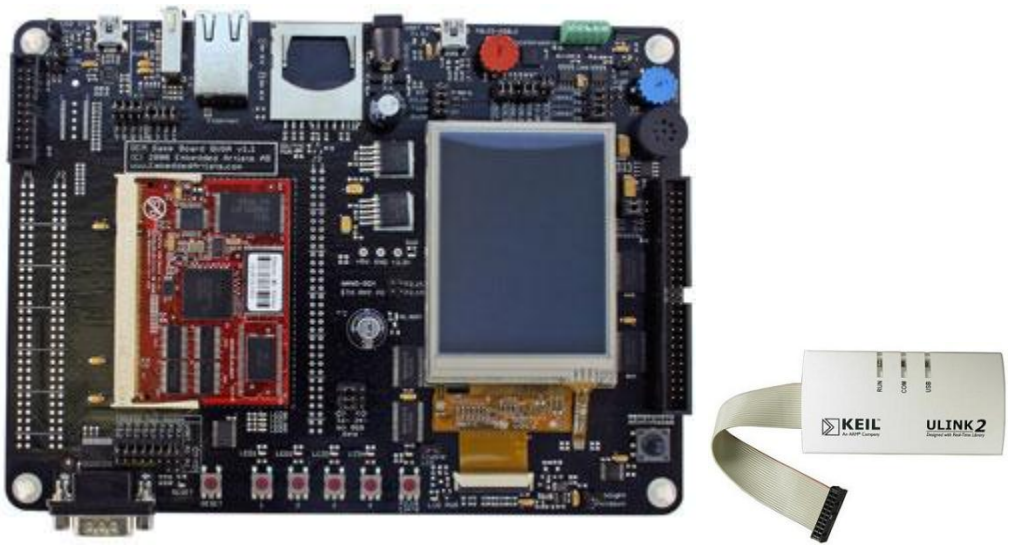

Figure 5: The LPC2478 Developers Kit from Embedded Artists. In conjunction with system software, application specific software and a touch of industrial design, you see what could be the humble beginnings of the next iPod competitor.

Real-Time in the iPod

Irrespective of the hardware and software platform chosen, an iPod-equivalent product would most likely be implemented as a soft real-time system. This choice would be made since it is highly desirable that the music play smoothly and without interruption, but it is not catastrophic if a deadline to provide a few milliseconds of audio is not met. This design choice is supported by the consistent application of an RTOS to the problem in the various development kits available.

One decomposition of a simplistic architecture for a portable music player is shown in Figure 6. This architecture uses three real-time tasks: the storage handler, the decoder and the device handler. The user interface is not considered a real-time task, but must operate quickly enough so that the user does not get frustrated by any delays.

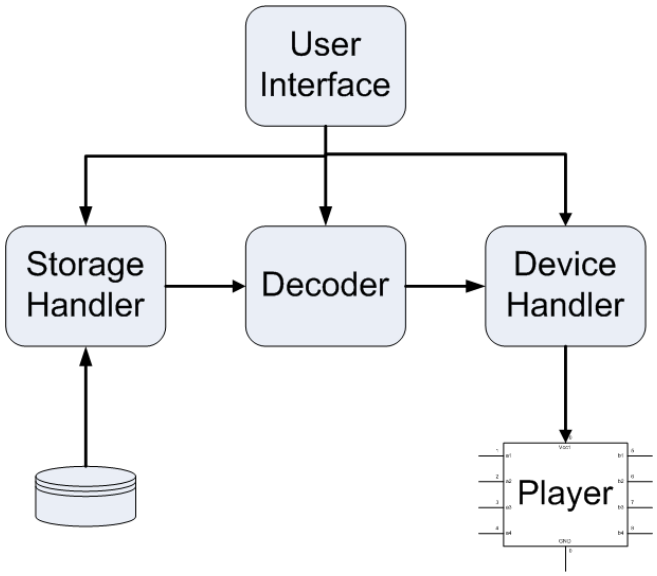

Figure 6: A simple architecture for a portable music player. The storage handler reads data from a storage medium and passes these data to a decoder. The decoder decompresses the data and passes them to the device handler which writes the decompressed audio data to the hardware.

The user of the portable music player will select a song to play via the user interface. The user interface orchestrates setting up the data path from the storage medium. Typically the audio data is stored in compressed form in "chunks" that correspond to a period of time. These chunks must be decompressed before being played, and so a task handles this decompression. Once decompressed, a chunk of music data may be sent to the player hardware and be "rendered" onto the user's headphones.

This simple architecture serves as a nice model to discuss some important concepts in real-time systems.

Definition 5: A real-time *task* is an executable entity of work that at a minimum is characterized by a worst case execution time and a time constraint [1].

Definition 6: A *job* is a specific instance of a task [1].

In this case, the storage handler, the decoder and the device handler would all be considered real-time tasks. For each "chunk" of audio data processed during the playing of a file, the data would need to be handled. The data is handled by a piece of code implementing the task, and each time a given task is scheduled and executed, it is called a job.

Each job has at least two scheduling criteria associated with it.

Definition 7: The job *release time* is the earliest point in time at which a real-time job becomes ready to, or is allowed to execute [1].

Definition 8: A job *deadline* is a point in time by which a real-time job must complete [1].

As described above, there are implications to not meeting a deadline. These implications lead to three classes of deadlines:

> Definition 9: A *hard* deadline means that it is vital for the safety of the system that this deadline is always met [1].
>
> Definition 10: A *firm* deadline means that a task should complete by the deadline or not execute at all. There is no value in completing a job after its deadline [1].
>
> Definition 11: A *soft* deadline means that it is desirable to finish execution of the job by the deadline, but no catastrophe occurs if there is a late completion [1].

There is a lot of freedom regarding how to arrange the functions in the simple player architecture. It is clear though, that some activity needs to read audio bits from storage, some activity needs to decode those bits and another activity needs to write those bits to the audio hardware. It is equally clear that this sequence of events needs to repeat at some relatively constant rate. Such a system is called *periodic* and there are a number of ways to schedule these systems. The simplest way does not require an operating system at all. This method is used quite often in military systems, and is called *Timeline Scheduling*.

Timeline Scheduling

Timeline scheduling consists in dividing the temporal axis into slices of equal length. The jobs are allocated a priori at a frequency derived from the application requirements [11]. A timer synchronizes activation of the jobs at the beginning of each time slice. In the general case, the time slice is called the *minor cycle* and the rate at which the entire schedule repeats is called the *major cycle*. For example, consider a system in which job A must execute every 25 milliseconds, job B must execute every 50 milliseconds and job C must execute every 100 milleseconds, the minor cycle is the greatest common divisor of the periods, or 25 milleseconds. The major cycle in this case is 100 milliseconds.

One possible scheduling solution is shown in Figure 7. Notice that job A is scheduled to happen every twenty-five milliseconds, job B is scheduled to happen every fifty milliseconds and job C is scheduled once every one hundred milliseconds. The minor cycle is twenty-five milliseconds and the major cycle is one-hundred milliseconds. If the areas of the blocks represent the worst-case time taken to perform each of the tasks, the scheduling solution shown in figure 7 is called *feasible*. Notice that if all three jobs were scheduled in one time slice, at least one of those jobs would miss its deadline. Such a schedule is called *non-feasible*.

The implementation of such a schedule could be as simple as a for loop as shown in Listing 1. The schedule is synchronized by a call to a sleep function that returns a phase number every twenty-five milliseconds. The phase number is the minor cycle within each major cycle. The loop then executes the appropriate job based on the phase number.

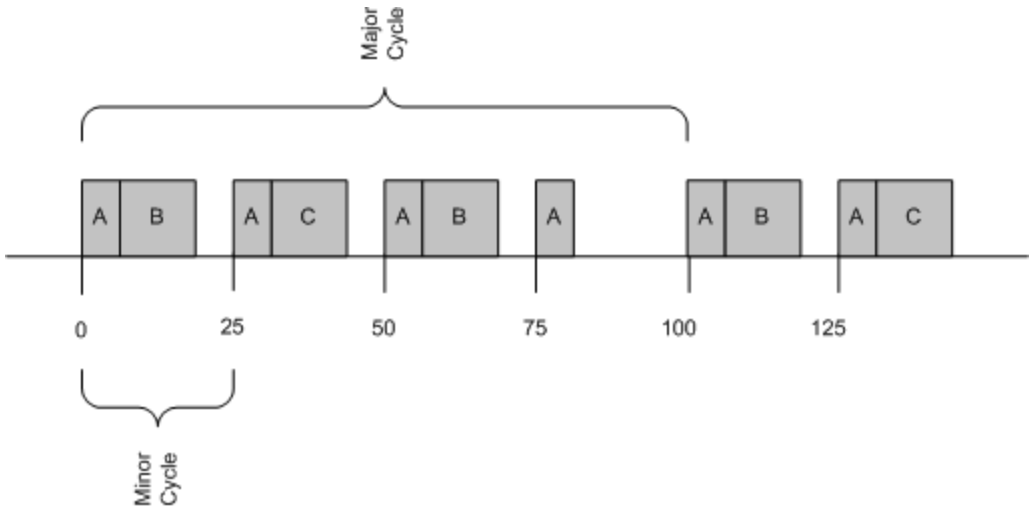

Figure 7: One possible scheduling solution for a hypothetical three-job periodic system.

```
for (;;) {
   phase = minor_cycle_sleep();
   execute_task_A();
   if (phase == 0 || phase == 2) execute_task_B();
   if (phase == 1) execute_task_C();
}
```

Listing 1: Pseudocode for implementing the schedule shown in Figure 7.

In the case of a portable audio player, the schedule will be primarily influenced by the audio data format. For example, many audio formats arrange data in chunks that correspond to portions of a timeline of the audio stream. In this case, the audio data in the storage medium may be arranged in variable length groups of bits that are expected to contain some number of milliseconds worth of uncompressed data. In order to keep the audio device provided with data, the minor cycle will then tend to be this chunk time or some multiple thereof. The simplest form of this system would be one in which all of the jobs above execute according to the same period as shown in Figure 8.

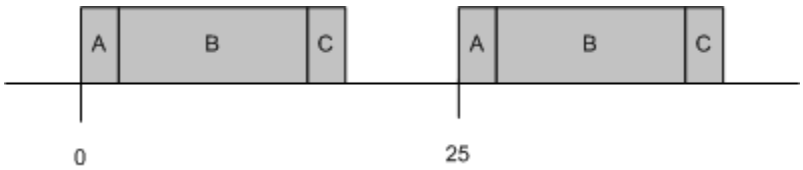

Figure 8: One possible scheduling solution for a hypothetical personal music player architecture.

In this case, both the major and minor cycles are 25 milliseconds. This would correspond to the chunk size of the audio format in question. Job A would be responsible for reading a chunk from the storage medium, job B would be responsible for decoding the compressed data and job C

would be responsible for writing the decoded data to the audio rendering device. The schedule would be feasible if the time to execute jobs A, B and C were less than 25 milliseconds.

At this point, computing power comes into the equation. The chosen processor must be able to execute the instructions required to decompress audio chunks (in the worst case) and during the major cycle if the schedule is to be feasible. Also, the data for the chunk must be retrievable (again in the worst case) during the cycle, and the system must be able to write the data to the device in that time.

Of course, in the real world these jobs are not executed at precisely 0, 25, … milliseconds. There will likely be some variations in the exact timing – often referred to as jitter. It is often useful to explore the causes of these variations and to make sure that they are not the cause of any kind of failure. There are several terms that one may encounter when speaking of jitter.

> Definition 12: The *relative release jitter* is the maximum deviation of the start time of two consecutive jobs [11].
>
> Definition 13: The *absolute release jitter* is the maximum deviation of the start time of all jobs [11].
>
> Definition 14: The *relative finishing jitter* is the maximum deviation of the finish time of two consecutive jobs [11].
>
> Definition 15: The *absolute finishing jitter* is the maximum deviation of the finish time of all jobs [11].

Systems can be constructed in ways that make jitter so small that it is not an important issue. For example, if the system is polled with a deterministic number of instructions executed in each loop, it is possible that jitter can be held to a level so that such a system could work well. Jitter could possibly be limited such a degree that only the asynchronous nature of the arrival of clock ticks contributes. Listing 2 shows a simplistic version of such a system.

```
for (;;) {
   minor_cycle_sleep();
   read_audio_chunk_from_storage();
   decompress_audio_chunk();
   write_audio_chunk_to_device();
}
```

Listing 2: Pseudocode for a simplistic audio player.

If interrupts are used, there will be issues with the asynchronous arrival of interrupt signals as well. It always takes some amount of time to dispatch interrupts in a real system, which will add to response latency. Timing may be affected by critical sections of code that occasionally disable interrupts, and execute in code that is asynchronous with respect to the critical interrupt paths. This would increase the amount of jitter in the code by the time spent in the critical

sections. If other, higher priority interrupts are happening in the system, these interrupts will also be arriving asynchronously. This will delay servicing of the interrupts in question and contribute to jitter by delaying for the duration of the higher level interrupt service routines. Any of these mechanisms could cause audible effects even if there is sufficient computing power to finish required processing well within the minor cycle time.

Figure 9 shows how a small absolute release jitter for job A can cause a gap in the audio. To minimize the effects of jitter, data is typically buffered in some way. In audio and video streams, this buffering is accomplished by delaying presentation of the audio stream through *preroll*. During preroll time the audio data is fetched and decompressed, but is not immediately rendered to the audio device. This mechanism assumes that the hardware playing the audio has the capability to play through at least two buffers without intervention by the CPU. Since the preroll buffering provides the hardware with more than one time quantum's worth of data, or more precisely enough data to play a quantum's worth of data plus enough data to play through the worst possible delay due to jitter at any time, the buffering masks the effects of the jitter (which is still present).

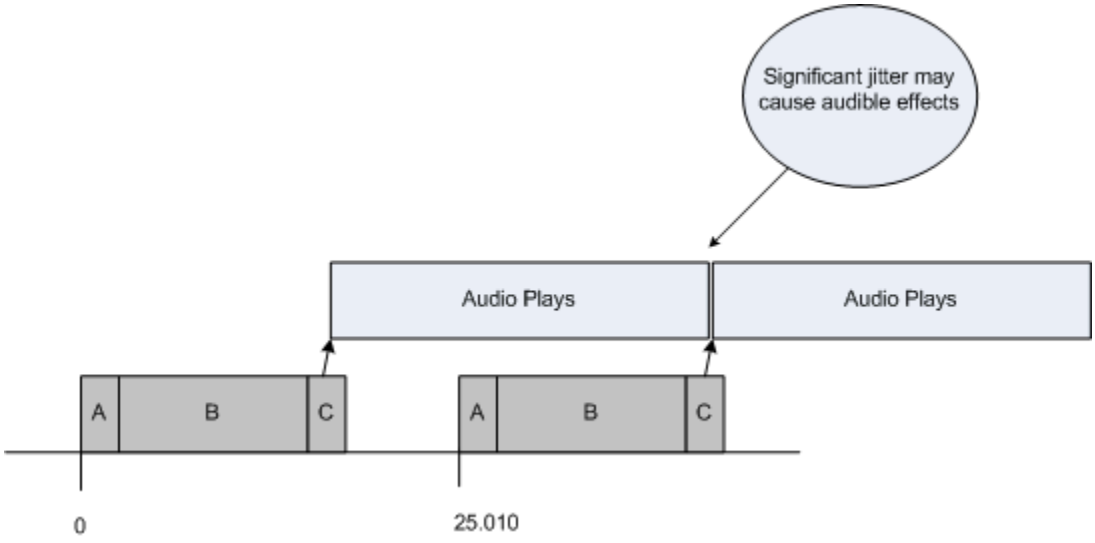

Figure 9: Jitter can affect even the simplest systems if not accounted for.

There are many ways to accomplish the buffering. Double-buffering is a common solution. In this approach, the hardware automatically switches between two dedicated buffers. For a more sophisticated example, the Local Area Network Controller for Ethernet (LANCE) implements a circular scatter-gather list in hardware with ownership bits to mediate access by the main CPU and the controller.

In any case, the presence of preroll is to accomplish buffering through some meachansim, and it introduces at least one non-periodic task into the mix – the stream start task. In order for the schedule to be feasible, all cycles must be capable of adding this preroll task. Figure 10 shows a preroll task added to the second cycle. In this case, there are two major cycles worth of buffered data present to absorb jitter.

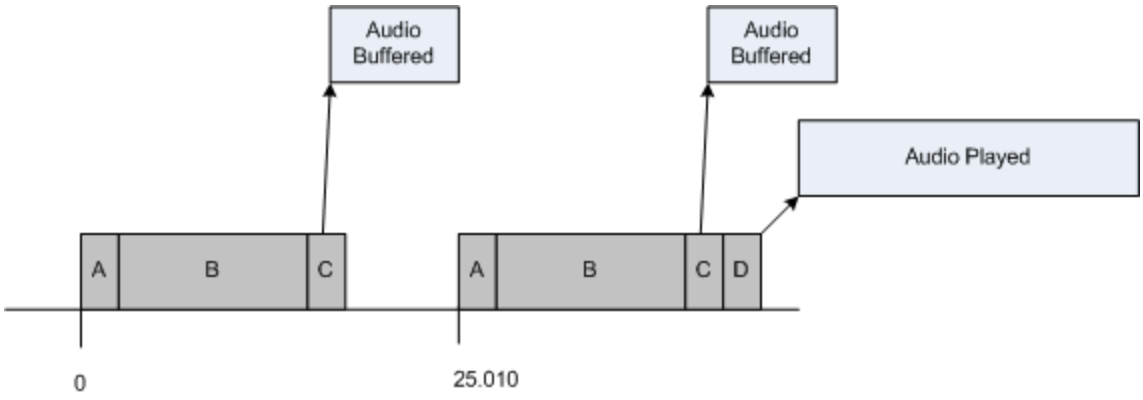

Figure 10: Buffering (preroll) can mitigate the effects of jitter on the audio playback system.

While the complexity of this design is manageable, there is one large piece that has been completely ignored in this analysis: the user interface. The entire system is architected around a timeline schedule with a fixed cycle time. In order to add a user interface task, it must execute in lock-step with the real-time. The user interface task must then be broken up into jobs that can fit into the timeline. This is illustrated in Figure 11.

This is obviously not a good solution since we are running the user interface as a real-time task and constraining all UI actions to fit within the cycle structure of the timeline. It is conceivable that two processors could be used, one dedicated to real-time timeline processing and the other to user interface code (is this why the PortalPlayer system on a chip has two processors). It is also possible to run the user interface as an application and use interrupts to drive the real-time activities. In this kind of system, the audio device interrupt could provide the driving timing signals. In practice, however, the preferred solution is to move toward a pre-emptive real time operating system (RTOS).

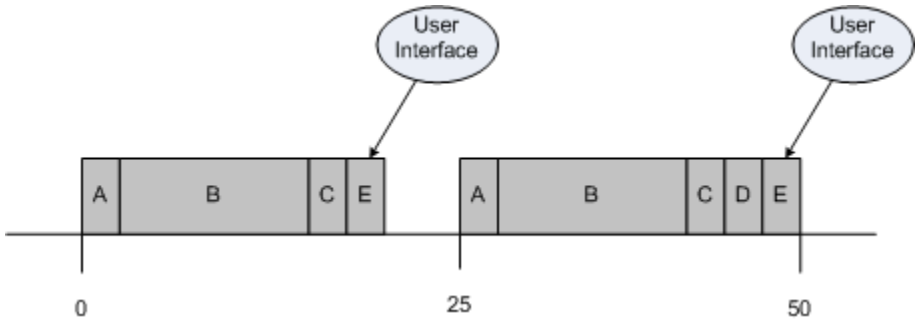

Figure 11: Shoehorning user interface code into the timeline schedule.

Scheduling problems are treated differently in a pre-emptive thread environment. In these environments, schedules are sometimes represented as an integer step function. In such a diagram, time is shown on the x-axis and the task number is shown on the y-axis. If no job is executing, the system is idle. This condition is described by a value of zero in the step function. Figure 12 shows a possible schedule for the timeline schedule of Figure 10.

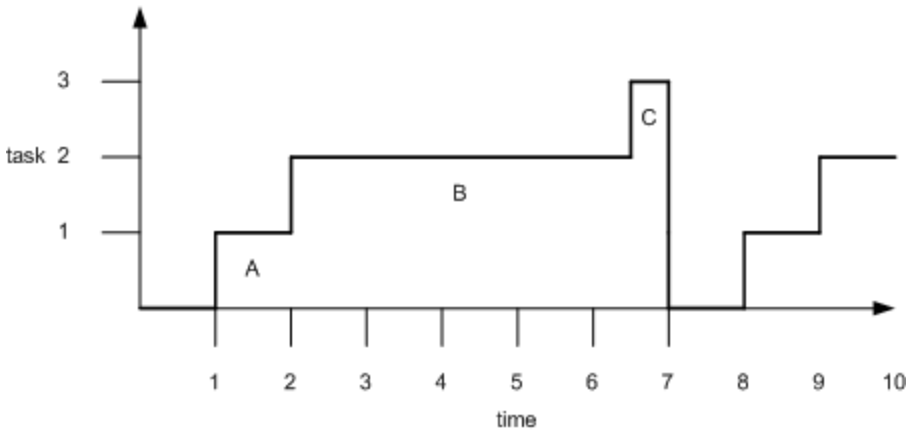

Figure 12: A schedule represented as an integer step function. Three tasks are executing. The system starts out idle from time zero to time one (the value of the step function is zero). A job corresponding to task one (the value of the step function is one) executes from time one to time two. At time two, a job corresponding to task two executes until time 6.5, at which a job corresponding to task three executes for one half time unit, etc.

In Figure 12, task one has been assigned to the Storage Handler, task two has been assigned to the Decoder and task three has been assigned to the Device Handler. The time equivalent to the first major cycle of Figure 10 is shown. The area marked A in Figure 12 corresponds to the job marked as A in Figure 10, etc. The step-function diagram equivalent to the timeline schedule of Figure 11 is shown in Figure 13.

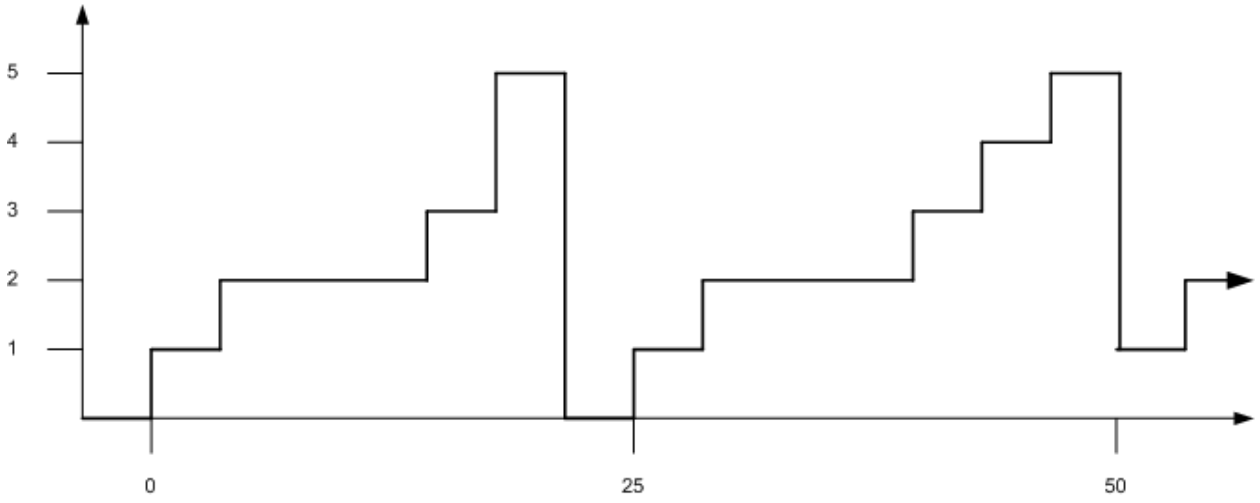

Figure 13: A schedule represented as an integer step function which is equivalent to the timeline schedule of Figure 11.

Another notation which is commonly seen is shown in Figure 14. Instead of showing the tasks as a step function, the diagram shows a two-dimensional timeline.

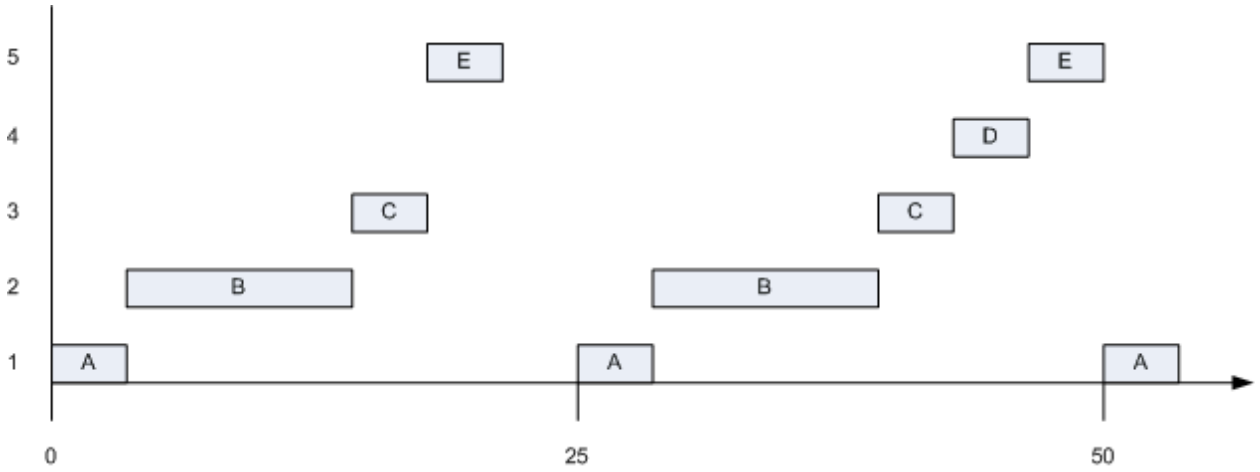

Figure 14: A schedule represented as a two-dimensional timeline which is equivalent to the timeline schedule of Figure 11 and the integer step-function schedule of Figure 12.

The differences between figures 11, 13 and 14 are purely notational, but one should be comfortable using any of the notations as there is really no standard.

Rate Monotonic Scheduling

When scheduling periodic tasks, the most common approach is called *Rate Monotonic* scheduling. This is a fixed priority method, the essence of which is that the tasks with the highest release rate are assigned the highest priority. The approach depends on the fact that pre-emption is possible. If we return to the schedule of Figure 7 and apply the rate monotonic (RM) technique, the resulting schedule is shown in Figure 15.
According to the RM algorithm, since job A executes at the highest rate, it is assigned the highest priority. Job B, with the next highest rate is assigned the second highest priority. Job C, with the lowest rate is assigned the lowest priority.

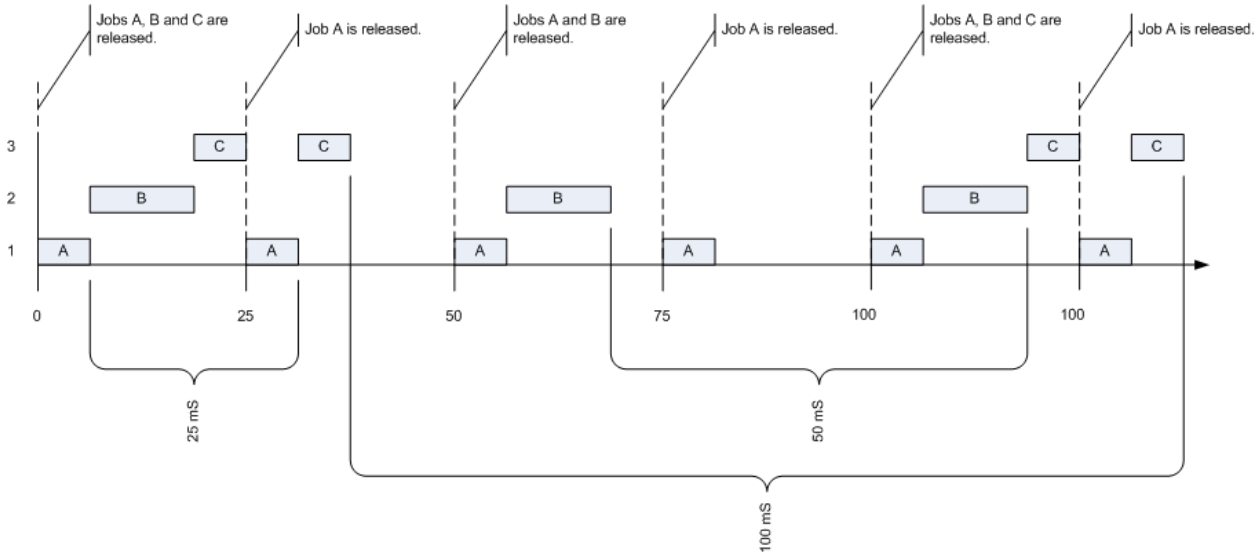

Figure 15: Applying the rate monotonic technique to the schedule from Figure 7.

In this case, the schedule is not put together offline and "by hand" as it was using the timeline approach. When using RM, the jobs are simply released at the desired rate. In Figure 15 notice that at time zero all three jobs are released simultaneously. Since job A has the highest priority it is executed first. When job A completes, the next higher priority job, job B, begins executing. When that job is complete, the lowest priority job, job C, begins executing. At 25 milliseconds the second instance of task A, job A is released again. Since this job has a higher priority than the currently executing job, job C is pre-empted and job A executes. When job A completes, job C is readied and begins executing until completion. Notice that job A repeats every 25 milliseconds, job B repeats every 50 milliseconds and job C repeats every 100 milliseconds. The equivalent of a major and minor cycle is clearly visible here.

It has been shown in the literature that rate monotonic scheduling is optimal in the sense that no other fixed priority algorithm can schedule a task set that cannot be scheduled rate monotonically [11]. In other words, if the schedule is feasible by an arbitrary priority assignment, it is feasible by RM. The RM algorithm is, however, sensitive to the processor utilization factor.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Here, C_i is the computation time – the time necessary to execute the job without interruption and T_i is the period of the job. The quantity $\frac{C_i}{T_i}$ is the fraction of processor time executing job i and so the above equation describes the processor utilization factor (from 0 to 1) for *n* jobs. There exists a maximum value for U below which a task set is schedulable and above which the task set is not schedulable. A convenient metric which accounts for several characteristics involved in guaranteeing the schedulability of a task set composed of n tasks is the least upper bound, labeled as U_{lub}.

$$U_{lub} = n\left(2^{\frac{1}{n}} - 1\right)$$

It has been shown that any task set with a processor utilization factor less than U_{lub} is schedulable by RM [11]. This quantity allows one to easily verify the schedulability of a given task set. Table 1 shows the values of U_{lub} for selected values of n.

n	U_{lub}
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
7	0.729
∞	0.693

Table 1: Selected values for the least upper bound value for various numbers of tasks.

For high values of n, U_{lub}converges to $U_{lub} = ln(2) \approx 0.693$.

To verify the schedulability of the task set shown in Figure 15, we need to know the computation times of the three jobs. If one assumes the computation time for job A is five milliseconds with a period of 25 milliseconds, the computation time for job B is 10 milliseconds with a period of 50 milliseconds, and the computation time for job C is ten milliseconds with a period of 100 milliseconds, we can calculate U.

$$U = \frac{5}{25} + \frac{10}{50} + \frac{10}{100} = \frac{1}{2}$$

Since $U = \frac{1}{2} \leq U_{lub(3)} = 0.780$ it is shown that the task set of Figure 15 is feasible. We could then determine how much processing we could allocate to another periodic job. If we assign job four to this processing task, the least upper bound becomes $U_{lub(4)} = 0.757$. The processor utilization factor for the existing jobs remains constant. If we use 100 milliseconds for the new job update interval, we can solve for the amount of processor time available for that job:

$$\frac{1}{2} + \frac{x}{100} \leq 0.757 \rightarrow x = 25.7\ ms$$

And so, if this additional job was, for example, a lower priority user interface update job, we would know that we could spend a maximum of 25.7 milliseconds every one hundred milliseconds executing a user interface while still being confident that our higher priority tasks would be schedulable by RM. Unlike the user interface added to the timeline scheduling example, the UI code in this case does not have to be broken up by hand into a very precise cycle structure and intermingled with the higher priority code. This results in a much greater amount of flexibility in the code and is the primary reason even relatively small systems are written using real time pre-emptive operating systems.

Of course, nothing in life is free, and this flexibility means that one must pay close attention to processor utilization and least upper bound calculations when using the rate monotonic technique.

The Large Hadron Collider

Although most real-time systems are relatively small embedded systems like the Apple iPod, there is really no fundamental size limit. The Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) is the largest machine in the world [9], and is also a real-time system.

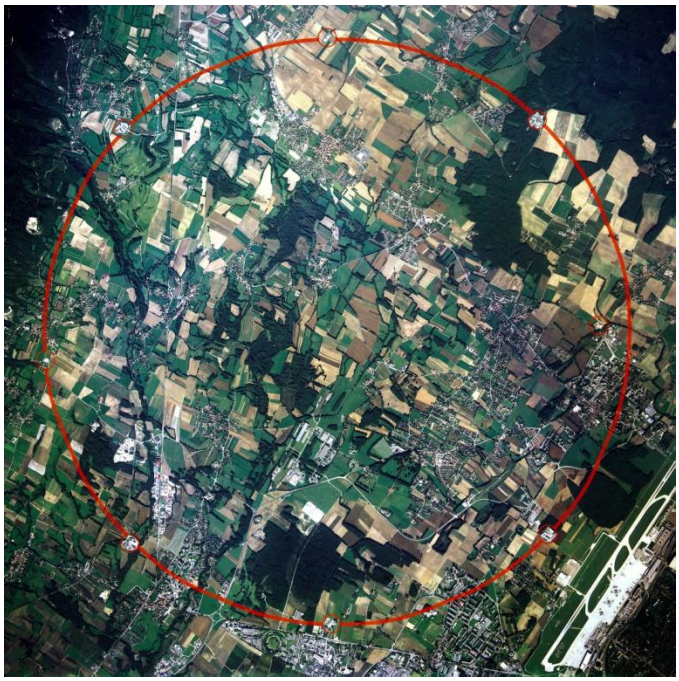

Figure 16: Aerial photograph of the Meyrin region outside Geneva, Switzerland with the position of the LHC main accelerator ring shown [10].

The LHC is a very high energy particle accelerator designed to explore the origin of mass, and to test hypotheses regarding dark matter and dark energy. The main accelerator ring is 100 meters underground, 27 kilometers in circumference and actually spans the borders of France and Switzerland near Geneva. Figure 16 shows an aerial view of the site for scale. The Geneva International Airport can be seen at the lower right side of the picture. This is indeed a large hadron-collider.

The LHC collides "bunches" of protons that travel in counter-rotating beams which move around the accelerator ring at a rate of 11,245 times per second. There are 2808 bunches in the main ring at any time in each direction, which allows for potential collisions occurring every 25 nanoseconds in windows of 180 picoseconds. The beams cross at so-called "experiments" which are arranged around the accelerator ring. Figure 17 shows a schematic of the LHC main ring with the four largest experiments, ATLAS, CMS, ALICE and LHC-b shown.

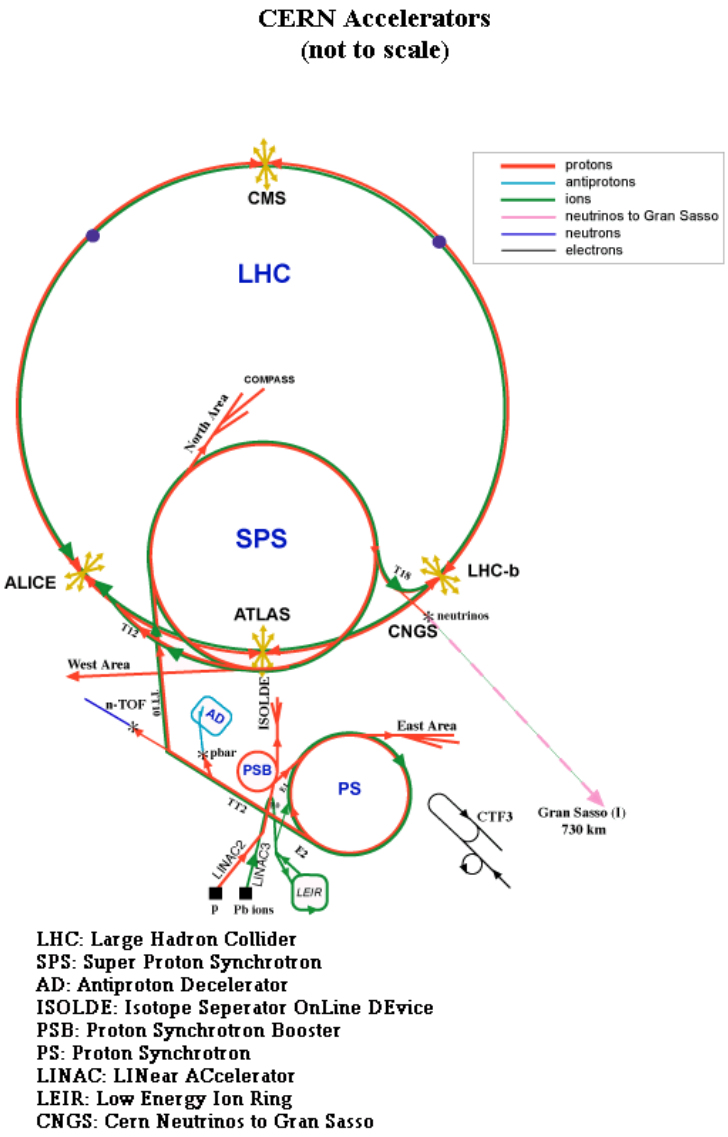

Figure 17: The hierarchy of accelerators that work together in the Large Hadron Collider. Taken from http://ab-dep-op-sps.web.cern.ch/ab-dep-op-sps/images/acceleratorssps.gif

The LHC is not entirely new, but builds on previous CERN experiments. At the bottom of figure 17 one can find the injection point labeled "P" for Protons. Protons (produced by ionizing hydrogen gas) are accelerated by a linear accelerator (LINAC2), fed into a circular accelerator called the Proton Synchrotron Booster (PSB) and then into the Proton Synchrotron (PS). The PS was actually commissioned in 1959, and was briefly the world's highest-energy accelerator at 28 billion electron volts (GeV). The output of the PS is fed into the Super Proton Synchrotron (SPS) which was a 450 GeV machine commissioned in 1976. Originally the 27 kilometer LHC ring was dedicated to the Large Electron-Positron (LEP) collider which was CERN's flagship accelerator starting in 1989 and eventually achieved collisions at energies of 209 GeV. The LEP was decommissioned in 2000 order to begin conversion to the LHC which is expected to achieve energies of 14 trillion electron volts (14 TeV) when it reaches design energies.

The largest experiment is the ATLAS detector. As shown in figure 18, the counter-rotating bunches of protons enter at each side, cross and collide at the center, at a point called the Interaction Point (IP). The detector is actually composed of several sub-detector modules that curround the IP. The Inner Detector is closest to the IP. Its function is to track charged particles and provide information about particle momentum. The inner detector is further decomposed into a Pixel Detector that provides over 80 million readout channels; a Semiconductor Tracker (SCT) with 6.2 million channels, and a Transient Radiation Tracker (TRT) with about 300,000 readout channels. The Inner Detector is, in turn, surrounded by electromagnetic and hadronic calorimeters (196,000 channels) and a Muon Spectrometer with 354,000 aliminum Drift Tubes arranged around the detector barrel and in endcap "big wheels" which can be seen in Figure 18. In layman's terms, ATLAS is essentially a 90 megapixel digital camera the size of a seven-story building that can take a picture every 25 nanoseconds. A man and woman are shown (in red) to the left of the detector barrel in the illustration below for scale.

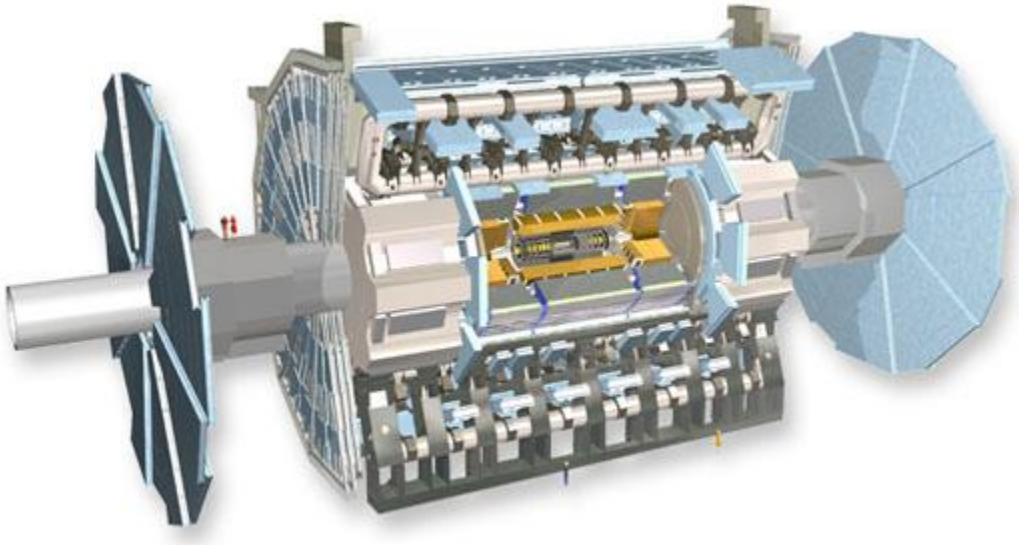

Figure 18: A cutaway illustration of the ATLAS experiment at the Large Hadron Collider. Note the illustrations of two people shown to the left of the main detector barrel for scale. Taken from http://www.atlas.ch/etours_exper/index.html

The scope of the real-time system design at the LHC can now be appreciated. Particle bunches must be injected into the PS and accelerated, then synchronously injected into the SPS and finally into the LHC. The detectors, separated by many miles, must be synchronized to the proton bunches. It is not possible to transmit and store the aggregate data from 90 million readout channels firing every 25 nanoseconds in the ATLAS detector alone, so the detectors must filter the data in real time, saving only interactions that are "of interest" to physicists operating the machine. This filtering reduces the flow of data from about a petabyte (a million gigabytes) of data per second down to a more manageable gigabyte per second. Even with this filtering in place, ATLAS will produce on the order of fifteen petabytes of physics data per year.

This data is consumed by a three tier computing grid. Tier-0 is located at CERN and distributes data to ten regional Tier-1 centers over ten-gigabit-per-second optical paths. For an idea of the

scale of the computing facilities, the Tier-1 computing facility in the United States is located at the Brookhaven National Laboratory and provides 3,000 CPUs and two million gigabytes of storage dedicated to processing ATLAS data. There are then over one hundred smaller Tier-2 centers which are geared to in-depth user analysis of the data.

There are a large number of hierarchical systems, implemented in hardware and software, which work together to keep the LHC operating and preventing it from becoming (literally) a smoking hole in the French countryside.

Real-Time at the LHC

The Beam Synchronous Timing System (BST) controls the timing of the proton bunches by distributing roughly 40 MHz bunch clocks over an optical network spread across the 27 kilometer ring. The orbiting bunches don't take care of themselves, so a hard real-time system called the Global Orbit Feedback System monitors and controls the LHC beam. There are 2112 Beam Position Monitors (BPM) feeding 70 BMP-Frontend PowerPC systems running LynxOS 4 for pre-processing the position data. A centralized Orbit Feedback Controller (OFC) talks to 50 Power Converter (PC) Gateways which, in turn, drive 530 correction magnets to control the beam. These devices are connected using a dedicated and redundant 32 gigabit per second Ethernet fiber optical network called the CERN Technical Network which provides thirty million packets per second throughput and quality of service guarantee on a hardware level. The OFC must handle about four million double floating-point multiplications per sample at 50 samples per second or about 400 MFLOPS with real-time constraints. The underlying system hosting the OFC is an HP Proliant computer running Scientific Linux Cern (SLC) version 5. SLC5 is a modification of Scientific Linux, which is assembled from Red Had Enterprise Linux (RHEL) sources and is freely available under GPL (see http://linux.web.cern.ch/linux/scientific5/ docs/ and https://www.scientificlinux.org/). The OFC uses the open-source realtime kernel from RHEL MRG (Messaging, Realtime and Grid – see http://www.redhat.com/mrg) to provide enhancements to make Linux better suited to environments with time constraints.

The BST system is a good example of a generic real-time control system as illustrated in Figure 19. The system must gather its inputs from a sensory system (the Beam Position Monitors), implement a control system (the Global Orbit Feedback system) and manipulate its actuation system (the Power Converter Gateways).

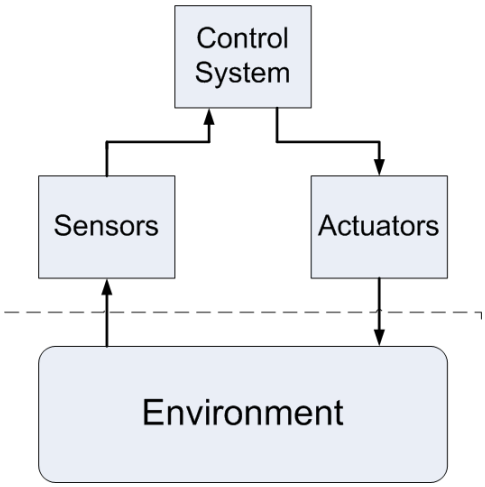

Figure 19: A block diagram of a generic control system.

The BST system, like the portable media player system described above, is another example of a periodic real-time system. A schedulability analysis of the OFC would not be significantly different than one performed on that portable music player. The control loop of the OFC must be executed fifty times per second and so the period is 20 milliseconds. If the computational requirements of the OFC main task are 400 MFLOPS, it is straightforward to determine the system processing requirements in a manner similar to the user interface task exercise for the media player shown above. In both cases it would involve calculating the processor utilization factors and the least upper bound metrics for the systems and solving for the desired quantities.

In contrast to the periodic nature of the Global Orbit Feedback system, the individual subcomponents of the ATLAS detector operate when particles are individually detected, much like a digital camera. The muon spectrometer is designed to detect subatomic particles, heavy cousins of the electron, called muons. The ATLAS Muon Spectrometer occupies the bulk of the ATLAS underground chamber and is illustrated in Figure 20.

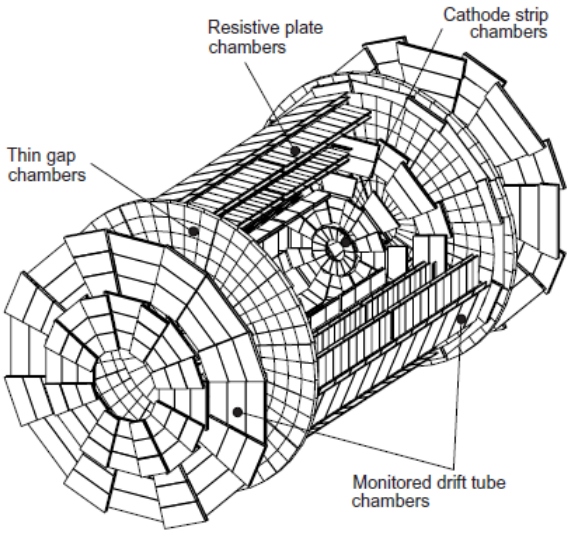

Figure 20: A three-dimensional view of the ATLAS Muon Spectrometer instrumentation.

The most visible components of the Muon Spectrometer are the Monitored Drift Tubes (MDT). There are 1,194 MDT chambers in the system, each composed of hundreds of individual drift tubes. A typical MTD chamber is shown in Figure 21.

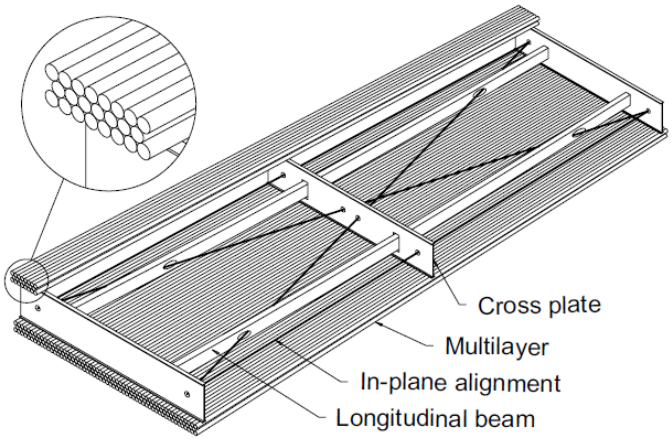

Figure 21: A cutaway diagram of an LHC MDT chamber showing the supporting structure and the individual gas-filled drift tubes used to detect muons.

The details about how the detection of the muons is actually done are quite interesting, but suffice it to say that when a muon flies through a drift tube, an electrical signal is generated that yields timing, position and deposited energy information.

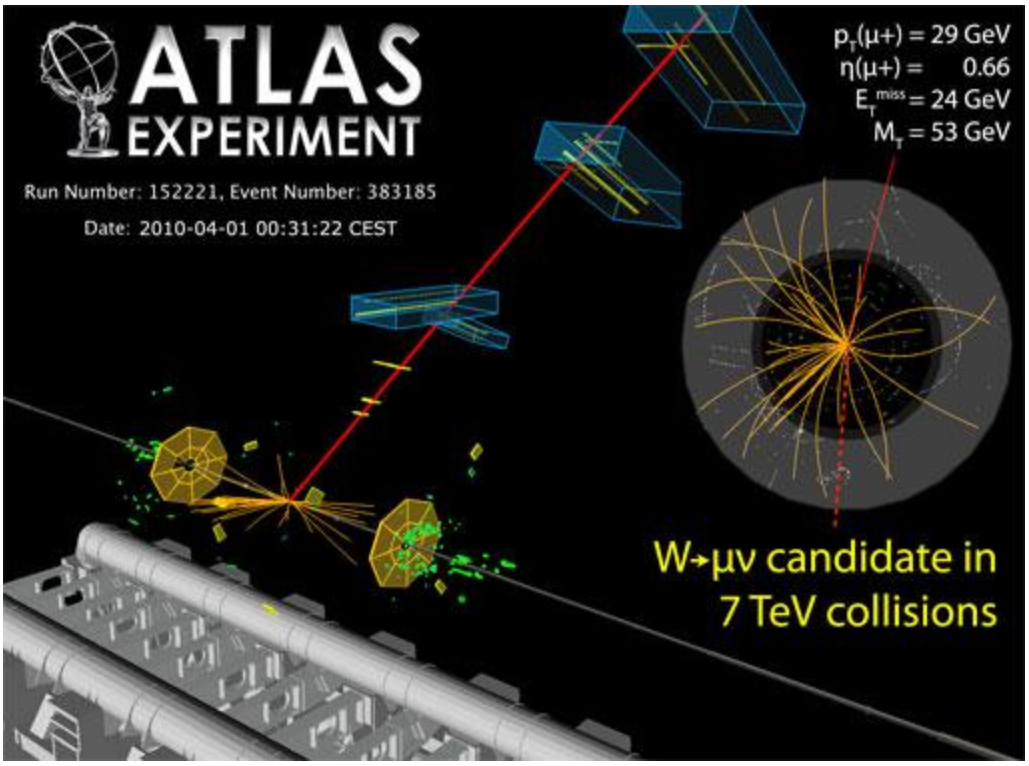

Figure 22: A reconstruction of a physics event in the ATLAS detector. The solid red line is a muon track, the path of which is determined by real-time data gathered from the Monitored Drift Tubes. [Taken from http://atlas-service-enews.web.cern.ch/atlas-service-enews/2010/news_10/news_W.php]

Figure 22 shows a reconstruction of a collision event in the ATLAS detector. The collision point is the origin of the red muon track and corresponds to the central point of the detector as shown in figure 18. The protons that collided in this reconstruction are baryons (bary- for heavy, relatively speaking). Protons have internal structure – they are each composed of three quarks. Quarks, in turn, have types, known as "flavors." Ordinary, everyday matter is made of "up" and "down" quarks. Two up quarks and a down quark are bound together to form a proton.

One possible result of a collision is that one of the quarks that make up one of the protons can decay into a "jet" of hadrons and a so-called W^+ boson. The $W^{\pm}$ bosons are the mediators (carriers) of the weak nuclear force and are extremely short-lived (and were first detected at CERN in 1983). The W can decay into a muon (μ) and a neutrino (ν) very quickly. The $W \rightarrow \mu\nu$ notation seen in figure 22 means that the W boson decays into a muon and a neutrino. Neutrinos are quite difficult to detect, and their existence is inferred from "missing" or undetected (transverse) energy (E_T^{miss} in the figure). The energy that went into the collision is known since we know the energy of the colliding protons (7 TeV). It cannot really be seen in figure 22, but the path of the muon is actually "bent" by a very intense magnetic field. From the bending of the muon track, one can discover the momentum and therefore the energy of the muon(s) escaping the detector. The muon spectrometer drift tubes actually hit by the outgoing muon can be seen in the blue boxes in figure 22.

It is important to account for all of the energy deposited in the various modules of the detector in order to infer a missing energy and therefore the presence of undetected neutrinos. The energy is detected by gathering information from the 90 million information channels of the detector. This happens in real time, possibly every 25 nanoseconds as protons in bunches collide at the interaction point of the detector.

There are a large number of detector sub-systems, but as an example, we can take a look at the muon spectrometer data acquisition system. The Monitored Drift Tubes described above are packaged into groups of twenty-four. These twenty-four drift tubes drive 24 Time to Digital Converters (TDCs). The value of the TDCs actually indirectly give the position within the drift tube at which the muon passed. Ninety-six TDC outputs are daisy-chained together onto a Front-End Link (FE-Link) channel over a serial line running at 80 megabits per second. Up to thirty-two FE-Links are connected to a NIKHEF MDT ReadOut Driver (NIMROD[1]). Four NIMRODs are packaged into Versa Module Europa (VME) Crates that are located on the detector. All in hardware, the NIMROD gathers the TDC information, adds information describing the source of the data, formats the acquired data and sends it via a 1.2 gigabyte per second fiber to what is called the ReadOut Buffer (ROB) system

As mentioned above, the volume of data must be significantly reduced since capturing a petabyte of physics data per second is impossible. This is accomplished by a three-level trigger system. The first level, the level-one trigger, controls whether or not the NIMROD forwards its data on to the ROB system. This triggering is done completely in hardware and takes information from the calorimeter (energy measurement) and muon spectrometer trigger chambers. Figure 23 shows a block diagram of the detector front-end electronics, the NIMROD (read-out driver) and the level one trigger hardware.

Level one triggers make decisions based on missing energy and activity in so-called Regions of Interest (RoI). For example, if the calorimeter detects a missing energy above some threshold it can generate a trigger signal. If, in the same event, the muon spectrometer also detects a muon flying out of a certain region of the detector, it can issue a trigger signal. The presence of both of these signals could indicate a possible $W \rightarrow \mu\nu$ event (cf. figure 22).

Since proton bunches may produce collisions every 25 nanoseconds, light travels about 25 feet in that time, and the detector is about the size of a seven story building, muons resulting from collisions of several bunches can be travelling in the detector at the same time. An additional task of the trigger is to correlate the muon tracks with the originating bunch so that corresponding calorimeter information can be combined.

The level one trigger reduces the data rates from 40 MHz (possible events every 25 nanoseconds) to about 75 KHz (possible interesting events every 13 microseconds). If the level one trigger decides it has observed a possibly interesting event it invokes the level two trigger. The level two trigger is implemented in software since the data rates have been reduced to a manageable level.

[1] Yes, physicists do exhibit a sense of humor when naming things.

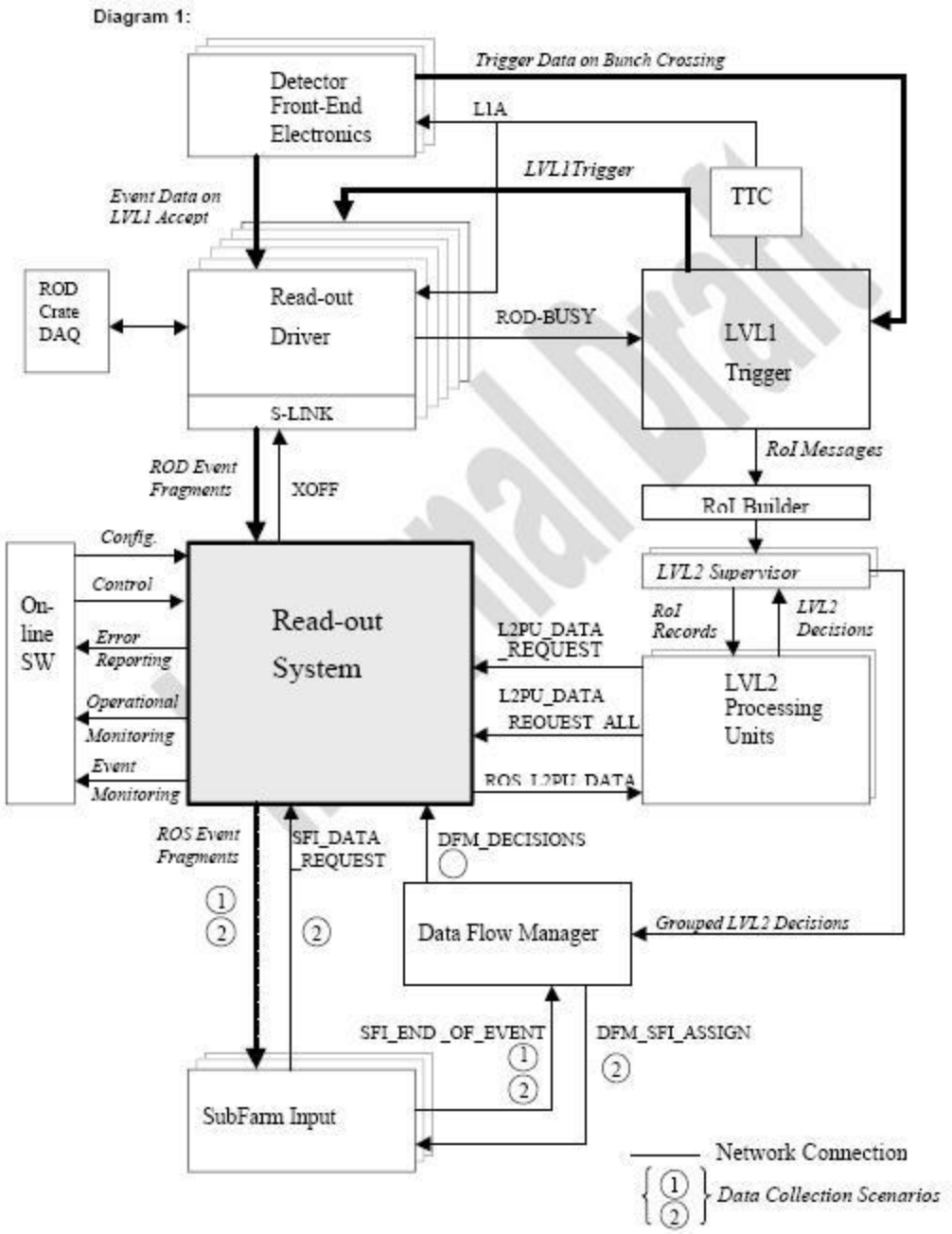

Figure 23: A block diagram of the muon spectrometer data acquisition (DAQ) system. The front-end electronics (FE-Links) are shown in the upper left. The NIMRODS are shown as Read-out Driver blocks. The 1.2 gigabit per second fiber connection is shown as an S-LINK.

The level two trigger software runs on standard PC hardware with eight Intel CPU cores per node. These PCs are mounted in racks located in the electronics cavern adjacent to the detector cavern. Each rack contains 31 machines. There are 17 full racks dedicated to level two triggers; and 28 racks are available as general purpose computing resource which can also be dedicated to level two trigger processing. That means that between 4,216 and 11,160 2.5 GHz CPU cores can be used in level two trigger processing [12]. These processors are shown in figure 22 as the LVL2 Processing Units. There is a centralized controller, called the LVL2 Supervisor that dispatches level two trigger jobs to the various CPU cores.

Each trigger processor node (8 cores) has 16 GB of memory available and two gigabit Ethernet network interfaces. The nodes run Scientific Linux CERN which has been described above. The level two trigger nodes run event-based software which performs quick/approximate physics reconstructions on certain regions of interest in order to determine if further analysis is indicated. The level two algorithms generally use about two to four percent of the available data, limiting the required bandwidth. The level two output rate is about 3,000 possibly interesting events per second.

Once a candidate event has passed level two trigger scrutiny, it is sent to the Event Filter. The Event Filter uses up to 62 racks of trigger processor nodes as described above. This applies as many as 15,376 processor cores to a more detailed analysis of the complete event data available in the read-out system [12]. The Event Filter is shown as the Sub-Farm Input in Figure 22, and a centralized Data Flow manager controls the dispatching of complete events to the Event Filter cores for processing.. The Event filter analysis results in a mere 200 events per second being tagged as interesting enough for further analysis. This results in the gigabyte per second of data which is stored for propagation to the three tier offline analysis grid.

The requirements for the real-time system of the Trigger and Data AcQuisition system (TDAQ) of the ATLAS experiment is different in a fundamental way from the real-time system of the Beam Synchronous Timing system, and the iPod for that matter. The BST system is fundamentally periodic, whereas the ATLAS TDAQ is aperiodic. The BST Orbit Feedback Controller (OFC) needs to wake up fifty times per second, like clockwork, and do the same thing. The ATLAS TDAQ needs to be ready to go at a moment's notice, but there is no deterministic way to know when it will need to wake up or for how long it might need to run. This is a fundamentally different scheduling problem.

Aperiodic Scheduling

When speaking of aperiodic scheduling, one sometimes encounters a systematic notation proposed by Graham et. al. The notation classifies the scheduling problem using a triple formatted according to $\langle \alpha | \beta | \gamma \rangle$. The first field, α, describes the machine environment (uniprocessor, multiprocessor, distributed, etc.). The second field, β, describes resource characteristics (preemptive, synchronous activations, independent, precedence constrained). The last field, γ, indicates the performance characteristic that is optimized, maximized, or minimized (maximum lateness, finishing times, etc.).

For example, $\langle 1|sync|L_{max}\rangle$ describes an aperiodic job scheduling algorithm that runs on a single processor machine. The $sync$ notation means that all jobs arrive synchronously, or at the same time. Since they all arrive at the same time, this algorithm is implicitly non-preemptive. No other constraints are imposed, and so all of the jobs are independent (no resources are shared and no precedence relationship). The goal of this algorithm is to minimize the maximum lateness. An optimal algorithm for this type of scheduling is called Earliest Due Date first (EDD). The algorithm is expressed by the following rule:

> *Jackson's Rule: Given a set of n independent jobs, any algorithm that executes the jobs in order of nondecreasing deadlines is optimal with respect to minimizing the maximum lateness* [11].

The class of aperiodic algorithms, $\langle 1|preem|L_{max}\rangle$ describes those that run on a single processor machine. The $preem$ notation means that dynamic job arrivals are expected (jobs may arrive at arbitrary times). This algorithm is explicitly preemptive. No other constraints are imposed, and so all of the jobs are independent (no resources are shared and no precedence relationship). The goal of this algorithm is to minimize the maximum lateness. An optimal algorithm for this type of scheduling is called Earliest Deadline First (EDF). The algorithm is expressed by the following rule:

> *Horn's Rule: Given a set of n independent jobs with arbitrary arrival times, any algorithm that at any instant executes the job with the earliest absolute deadline among all of the ready jobs is optimal with respect to minimizing the maximum lateness* [11].

Just as Rate Monotonic scheduling is the most common algorithm for addressing periodic job sets, Earliest Deadline first is the most common algorithm used for addressing aperiodic job sets. The algorithm can be easily generalized to multiprocessor systems and so can be found in a large number of real-time operating systems.

Consider a job set of five jobs with various release times a_i, computation times C_i, and deadlines d_i as seen in Table 2.

	Job 1	**Job 2**	**Job 3**	**Job 4**	**Job 5**
Release time	0	0	2	3	6
Computation time	1	2	2	2	2
Deadline	2	5	4	10	9

Table 2: Job parameters for an example execution of an EDF schedule

The schedule for the job set is shown in figure 24. Job one and job two are released at the same time, time zero. Job one has the earlier deadline, so it is executed first. Job one executes for one time unit and completes ahead of its deadline. Job two is ready at time one, having been released at time zero. According to the EDF algorithm it begins executing at time one. Job three is released at time two. At this time, job two is running. According to the EDF algorithm, the scheduler needs to evaluate the deadline at each instant, so it notices that job three has a deadline before job two. Since this is the case, job two is preempted and job three begins to run. At time three, job four is released. Since job four has a deadline after the currently running job, job three continues to execute until it finishes right at its deadline. At this point, the scheduler has two ready jobs, job two and job four. Job two has the earlier deadline so it is run to completion and finishes at its deadline.

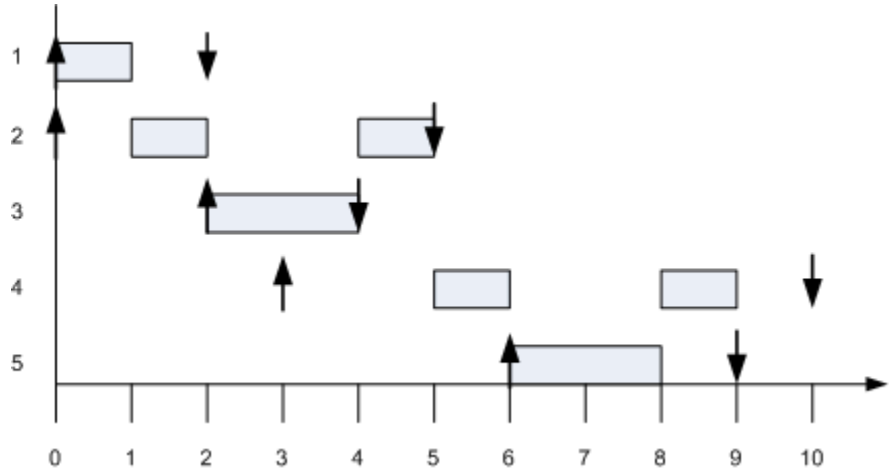

Figure 24: A schedule for an example job set as defined in Table 1. Release times are indicated with an up-arrow. Deadlines are indicated with a down-arrow. Execution of a task is indicated with shaded box.

At time five, the scheduler only has one ready job, so it begins executing job four. At time six, a new job, job five, is released. Job five has an earlier deadline than job four, so job four is pre-empted and job five begins executing. Job five completes at time eight, before its deadline. At time eight, the scheduler only has one ready job so it runs job four to completion, and that job finishes ahead of its deadline.

In order to make any guarantees about meeting a deadline, an aperiodic job scheduling algorithm needs an accurate estimate of the worst case computation time of a task. This was also true in the Rate Monotonic case, but the computation time was not attached directly to the job. The offline RM analysis used that information to determine if a schedule was feasible. A real time operating system using EDF is going to need explicit computation times and deadlines in order to guarantee the feasibility of a schedule.

Real-Time Linux?

The ATLAS experiment uses Scientific Linux CERN 5 for most of its operating systems. Recall that SLC 5 is based on Red Hat Enerprise Linux (RHEL) with the MRG (Messaging, Realtime and Grid) extensions. This version of Linux is advertised as a real-time version of Linux.

Process creation, thread creation and scheduling, however, all follow the usual Posix APIs. This means that one has the ability to adjust process and thread priorities, and with MRG, a user thread can actually take on a higher priority than kernel threads. One can adjust the scheduler policy to round robin (SCHED_RR) or first-in-first-out (SCHED_FIFO) , but look as hard as you may, there is no place to put a computation time estimate or a deadline. The Linux scheduler is not really a real-time scheduler, even in the case of real-time MRG Linux.

What has been done in the MRG kernel is that it has been made internally pre-emptive. This allows what are typically long-running, lower priority kernel operations to be pre-empted by the kernel itself. This reduces latencies and makes response times much more predictable. Additionally, MRG provides support for priority inheritance, to avoid problems with priority inversion; and fine granularity timers are introduced. This makes SLC 5 much more nimble and predictable.

Recall that the Beam Synchronous Timing (BST) system is a hard real-time system that controls the orbiting bunches of protons. If the 525 megajoule beam gets out of control, it would indeed be a catastrophic failure. The schedule on the centralized Orbit Feedback Controller (OFC) can be determined to be feasible by Rate Monotonic analysis. The 2112 Beam Position Monitors are running the LynxOS system from LynuxWorks. LynxOS is another Posix-compatible operating system. Since Posix specifies the APIs for the system, there is again no place to put a computation time and a deadline. LynxOS therefore does not support EDF. It turns out that the hard real-time guarantees are enforced by a hardware-based monitoring system called the LHC Beam Interlock System (LBIS) that extracts the beam into a water cooled beam dump to safely shut down the system. In systems such as this, worst-case behavior is compensated for by buffering, and catastrophic situations are detected by external agents – the hard real-time parts are implemented in hardware.

This is actually a common scenario. The underlying real-time operating system is a commodity OS supporting common Posix interfaces and therefore a soft real-time system. The OS may be optimized to reduce latency, but is not capable of meeting timing requirements. The system may be designed so that an external agent (a hardware system in the case of the LHC) monitors the condition of the system and takes some corrective action if a problem is encountered. This is in essencey no different than using a hardware-based I/O system in a general-purpose computer to provide functionality that the CPU is incapable of performing, or that would severely tax the CPU if it could.

RTLinux

Sometimes it is not good enough to detect that a catastrophic situation is about to occur and abort gracefully. For example, it is not a very good plan to design a system that will detect that a

rocket is about to explode since a deadline was missed. It is much better to guarantee a priori that the deadline will not be missed and the rocket will not explode.

If a combination of Posix compatibility and hard real-time is required, an interesting approach is taken by RTLinux. In this system, a small hard real-time kernel runs Linux as a low priority task. In fact, the Linux task is the idle task for the real-time kernel.

In RTLinux the low-level scheduler is modular, and a scheduler that supports EDF is available, as is a scheduler optimized for Rate Monotonic problems [13].

Summary

The design of real-time systems is a complex endeavor. The combinations of possible hardware and software components in the architecture, along with the different types of constraints imposed by the design lead to an essentially infinite number of solutions. There are many operating systems that advertise themselves as real-time, however, in general, most of these are soft real-time systems that attempt to maximize determinism and minimize latencies with various degrees of success.

Often product requirements will stress the importance of available API such as Posix compatibility for ease of integration. Since Unix was originally a general-purpose time-sharing system, Posix compatibility actually prevents the adoption of real-time features unless other out-of-band mechanisms are adopted. These can be over-specifying or over-building the system so that the soft system can *always* accommodate worst-case behavior thereby guaranteeing performance. Guarantees can be enforced in soft real-time systems using external agents. It is, of course, possible to build a hard real-time system, but choices of pre-existing standard solutions are somewhat limited.

The underlying hardware and software design also cannot stand alone, however. Constraints coming from product design, industrial design, manufacturing analysis and costing, and time-to-market issues can and will have a large impact on the final design.

Design is choice. Most often, the struggle is not that there is no way to solve a design problem, but that there are too many ways that apparently solve the problem. One must know which collection of solutions solves the problem in a satisfactory manner and why. The need to choose propagates from the largest architectural levels down to the smallest unit of code [15].

It is relatively easy to design a hardware or software system. It is much harder to design a system that works and meets its requirements; and it is very difficult indeed to produce a product that comes close to an optimal solution given often conflicting requirements. In order to come close to building a good product, people in all of the development roles must understand the concepts of real-time and work together to make sure the system comes together.

The Development Roles

In order to build a system that is even remotely optimal, all of the people in the development organization need to work together and understand what is being built and why. Real-time systems can sometimes be counter-intuitive to design, build and test. If building one, it is important for all of the people in the organization to have an understanding of real-time systems in order to ensure their piece of the puzzle is optimal by some agreed upon measure.

The Program Manager

Traditionally, a program is a group of related projects that move together to a common goal. In this management decomposition, the program manager provides management support for a group of projects. A traditional program manager manages the group of projects more than people.

In the software industry, a different approach is taken. Typically the program is viewed as a piece of software or a software system. The program manager is viewed as a technical leader, rather than a manager. The program manager is responsible for determining the requirements for the software, a feature list and the high-level design. The program manager then shepherds the project through detailed design, implementation, testing, release, and eventually through ongoing maintenance and support. In this pattern, communication skills are extremely important since the process of gathering and communicating requirements requires a great deal of cross-group and cross-functional communication.

When dealing with real-time projects, a good understanding of the implication of requirements to the resulting design is critical. Overspecifying the requirements can have tremendous negative impacts on the design. One has to be careful not to specify such things as, "the system must run the RTLinux hard real-time operating system" unless there is a very good reason to do so.

Consider the CERN example above. The LHC Beam Synchronous Timing system is a hard real time system, with responsibility to avoid truly catastrophic failures by monitoring the accelerator and dumping the beams if they get out of control in any way. There is, though, no hard real-time operating system in sight. What is used is a standard operating system (SLC 5) that is used in the tens of thousands of instances at CERN and has all kinds of support for remote system administration, updating, etc. There are hundreds, if not thousands of software engineers there with much experience with SLC 5. There is a strong requirement for having a system that is easily administered and there is a strong SLC 5 knowledge base at CERN. The requirement for the BST system is really that it can detect and prevent a catastrophic accident. By carefully specifying the "true" requirements of the BST, and not arbitrarily specifying a type of OS, one has the flexibility to come up with a solution that is more optimal for the organization as a whole by allowing a standard CERN operating system.

In the ATLAS trigger and data acquisition system (TDAQ), it might have been tempting to specify a hard real-time system in order to get an earliest deadline first scheduler and be able to have a deterministic system that could do the required filtering. Upon reflection, a CERN program manager or architect might have realized that although EDF would be nice, the triggering problem is by its nature a soft real-time problem. It would be nice to get as many trigger events as possible without losing them, but it is not catastrophic if a possibly interesting

event slips by unsaved. The real, basic requirement is that on average, the level two trigger is able to respond fast enough to analyze event fragments and produce a level three trigger. By introducing buffering and compute farms, CERN was able to meet the actual core requirement of filtering events while at the same time re-using existing physics analysis code and frameworks, and taking advantage of CERN standard operating systems with their concordant benefits.

In the case of the Apple iPod, Steve Jobs had made an observation that existing media players had clunky, hard-to-use user interfaces. He was approached with the idea of a media store, and went with the idea. The requirements were really to have a slick user interface, an "insanely great" industrial design and to get it to market within a year to beat the competition. Essentially nothing was specified, and this allowed Apple to go out and cobble together a solution which met their needs. What they got was a soft real time system with a pre-emptive operating system using round robin and FIFO scheduling (the same as in Linux).

The program manager should always be looking to sort the actual hard requirements from arbitrary a priori or ad hoc decisions. These requirements will be used by either the program manager (acting as a software architect) or a dedicated architect to come up with the actual high-level design. If a separate person with architectural responsibility is involved, the program manager must communicate extremely well with that person to ensure that a developing design does reflect the needs of the program. The program manager needs to communicate precise requirements to the architect, fixing only the design points that are required to be fixed, but leaving as many degrees of freedom as possible.

The Architect

The architecture of a system is the fundamental organization of a system embodied in its components, their relationship to each other and to the environment; and the principles guiding its design and evolution [14].

The architect would then define technical standards, platforms and tools, and subdivide the project (or program if you wish) into manageable pieces. The architect will be expected to understand the overall system behavior, define the components of the system and understand the interactions and inter-dependencies of the components.

The architect would be the person who, in the case of the ATLAS TDAQ, would understand that there needs to be a level one trigger operating in hardware since no reasonable computer system would be able to keep up with the data rates. She would understand that the read-out system needs to buffer detector data and trigger a soft real-time system for the level two trigger, which in-turn analyzes events and triggers a level three system for further online analysis. The architect of such a system would define the amount of buffering required in the read-out system and the number and capacity of the compute farm nodes that would be required to support the desired data rates.

In the case of something like the Apple iPod, an architect would be responsible for ensuring that the acquired pieces of the system were capable of working together to achieve the goals set by the program manager.

An architect should have a good knowledge of the underlying problem space in order to ensure that she is not asking the impossible of the system. The precise requirements of each component need to be extrapolated from the initial requirements and a high-level design constructed. The high level design and requirements must be communicated to the developers, again avoiding overspecifying the design. As with the program manager, the architect needs to communicate precise requirements to the developers, fixing only the design points that are required to be fixed (by the high-level requirements and her high-level design and standards), leaving as many degrees of freedom to the developers as possible.

The Developer

It is the developer's job to create the code to actually implement the real-time systems. The developer will write the code to implement the tasks that make up the job sets implied by the architect's high-level designs.

The developers may be the ones to do the Rate Monotonic analysis on the job set specified by the architect, taking into account the processor power available on the computers the architect has specified. The developers may be the ones to look and see if aperiodic task set schedules are feasible on the system given the real interrupt and scheduler latencies. The developers may be the ones to analyze the IO systems of the machines to make sure they can support the required rates.

The developers are responsible for the low level design of the system, but are also responsible for applying backpressure up the design chain in case impossible demands are made on the system. For example, if in the case of a hypothetical particle accelerator, the architect decided that an Orbit Feedback Controller running Scientific Linux CERN 5 would be responsible for deciding to dump the circulating beam and avoid catastrophic failures, the developer needs to remind the architect that even though SLC 5 is advertised as real-time, it really is not! The OFC may be able to handle the beam in most circumstances, but if a hard interlock is required, the architect had better rely on something else (hardware) because SLC 5 is not up to the job.

The developer should, of course, have a deep knowledge of real-time systems, but the more the developer can be aware of higher level requirements that are driving the system design, the better. Thus, the developer needs to communicate well with architect to ensure that nothing slips through the cracks and the system is meeting all of its requirements.

The developer is also the person who is most intimately involved and knowledgeable about both the strengths and weaknesses of the overall design. This puts him or her in the best position to communicate to the tester what the perceived weak points of the system are, or where concerns may lie.

The Tester

It is the tester's ultimate responsibility to make sure that the system does what it is supposed to – that it successfully fulfils its requirements. It is very important therefore, that requirements be testable. We have most likely all heard stories about development at Apple Computer. Essentially, the requirement comes down from Steve Jobs that Apple produce an "insanely great"

product. Apple industrial designers go off and design what they think is an "insanely great" product only to have Mr. Jobs declare that the result is not insanely great, it is the stupidest thing he has ever seen. It is hard to design tests in this world.

Especially when considering real-time systems, the criteria for a successful system are difficult to quantify precisely. Consider the iPod. It is a soft real-time system, so there may be times when the played audio breaks up due to a missed deadline. Is there a hard limit for how many times a personal music player may "skip a beat" during a day. Is this a hard limit or is it something someone just made up. What happens if the number is exceeded? Do you just change the number or are you forced to go back and re-evaluate the entire design. If your criterion is that the audio sounds "good enough," who decides what "good enough" means?

The testing phase is where the design meets reality. The more carefully a project is specified in terms of actual performance, the better the design can be and the easier it will be to validate the design. Testers need to take in information from all of the other members of the development team regarding the design and the requirements and reply, again to all members of the team with information on how well the design meets requirements.

Real-Time Oddities

Real-time systems can sometimes throw curves that unsuspecting engineers are completely unprepared for. Real-time computing is not equivalent to fast computing. It turns out that an increase in computational power or a reduction in latency does not always translate into an improvement in the performance of a job set. Anomalies such as these are called Richard's anomalies [11]. Architects and developers should be aware of them so they can avoid them, testers should be aware of them so they can test for them and project managers should be aware of them so they do not always press for simply faster, better hardware. Project managers should also be prepared to try and explain these situations to external stake holders.

Richard's anomalies occur in jobs sets with precedence relations in multiprocessor environments. This is most likely the environment in which a modern system will be expected to operate. The following theorem summarizes the most important anomalies:

> *Graham's Theorem: If a job set is optimally scheduled on a multiprocessor with some priority assignment, a fixed number of processors, fixed execution times and precedence constraints, then increasing the number of processors, reducing execution times, or weakening the precedence relations can increase the schedule length.*

This is quite counter-intuitive. It says that adding a processor or optimizing your routines can actually make things worse!

	Job 1	**Job 2**	**Job 3**	**Job 4**	**Job 5**	**Job 6**	**Job 7**	**Job 8**	**Job 9**
Computation	3	2	2	2	4	4	4	4	9

Table 3: Computation times for an example three-processor schedule.

Consider a job set with precedence relations as shown in figure 25. The jobs are illustrated as circles and the precedence relations are shown as arrows. For example, at the top of the figure, notice that job nine (J9) depends on the completion of job one (J1). Jobs five, six, seven and eight depend on the completion of job four; and jobs two and three are independent. The computation time of these of these jobs is given in table 3.

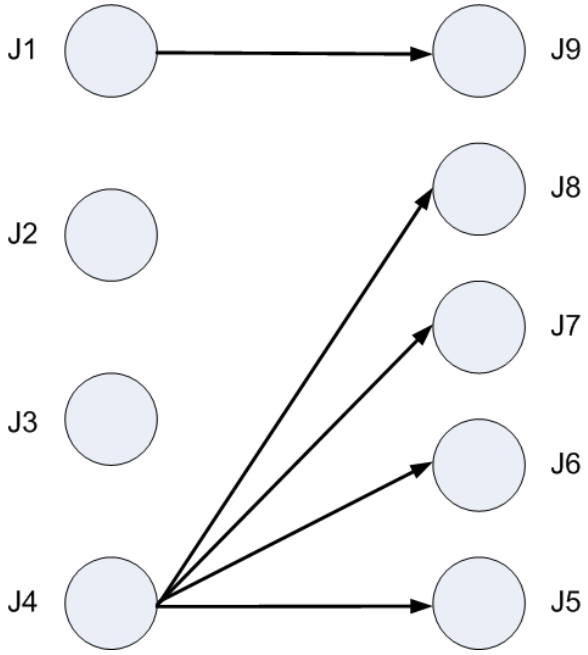

Figure 25: A precedence relationship used to illustrate a Richard's anomaly where an increase in performance results in an increase in schedule length.

Using the precedence relationships shown in figure 25 and the computation times shown in table 3, we can fairly easily come up with an optimal schedule for this job set. This is shown in Figure 26. Assume all of the jobs are released at time zero. Based on the precedence relations, jobs one, two, three and four are ready at time zero. The scheduler assigns the first three jobs to the first three processors.

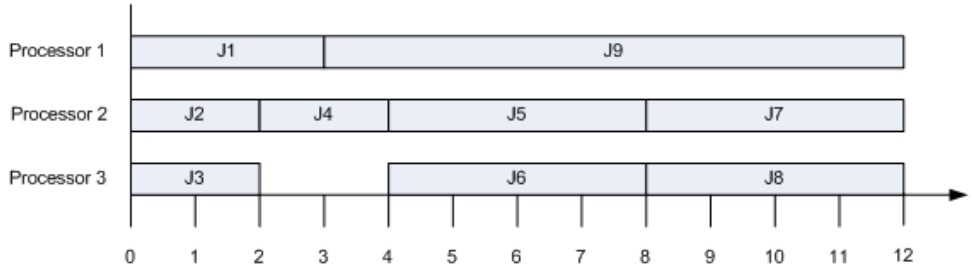

Figure 26: An optimal three processor schedule for the job set shown in figure 25.

As shown, jobs two and three finish at the same time, leaving two processors free to do work. Since job four is ready, it is assigned to processor two and begins executing. At this point,

though, the remaining jobs are waiting for either jobs one or four to complete, so there are no other ready jobs at this time. When job one finishes, processors one and three are free. The precedence graph indicates that job nine is ready, so it is assigned to processor one and begins executing. There are no further ready jobs so processor three remains idle. At time four, job four completes. This makes jobs five through eight ready based on the precedence graph. The scheduler then assigns job five to processor two and job six to processor three. All three processors are busy until jobs five and six finish at time eight. Jobs seven and eight are assigned to the free processors and jobs nine, seven and eight continue executing until time twelve when the job set is complete.

Let's assume that someone has the bright idea that adding another processor to the system will make it faster. Figure 27 shows what might happen when this is done. According to the precedence graph, jobs one, two, three and four are ready to execute at time zero so they are all dispatched to processors and begin executing. Jobs two, three and four finish executing at time two. Since job four has completed at time two, jobs five, six, seven and eight become ready to run and jobs five six and seven are run on the free processors. There is one job left, job nine which became ready after job one completed at time three. It is run on the first free processor at time six and completes at time fifteen.

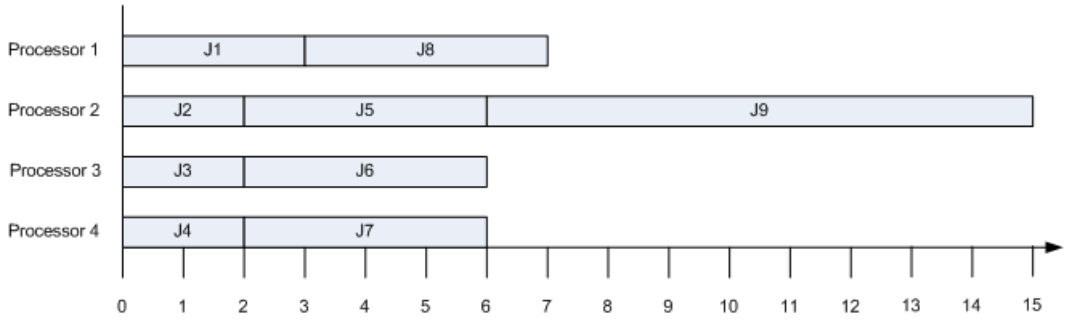

Figure 27: An optimal four processor schedule for the job set shown in figure 25. It takes longer to execute with an additional processor.

By adding another processor to the machine, it has resulted in the schedule taking three time units longer to execute!

Let's assume our bright engineer decides that since adding a processor made the system slower, he should undo that. Seeking better performance in other ways, he decides to replace all three processors with faster processors. Surely that will make things better, he thinks. Let's take a look at what might happen if the computation time for each task is reduced by one time unit. The results are shown in figure 28.

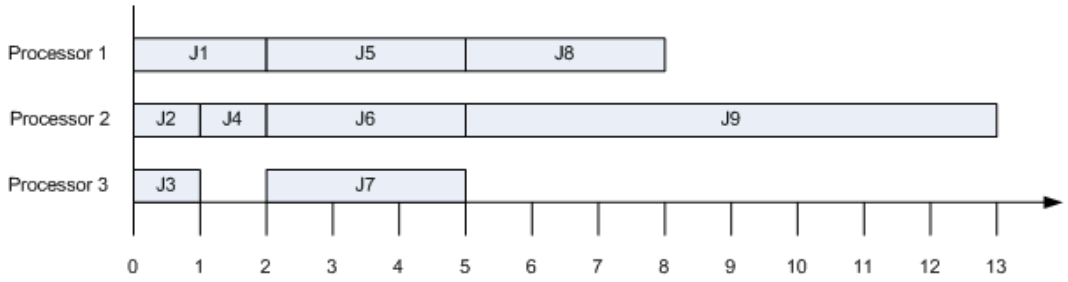

Figure 28: An optimal three schedule for the job set shown in figure 25. In this case, the computation time of each task is reduced by one unit of time. The result is that it takes longer to execute with faster processors.

Again, just as in figure 26, the first three jobs are immediately dispatched. Job two finishes and job four replaces it on processor two just as in figure 26. Now, jobs one and four finish at the same time and so there are three free processors at time two. Since jobs one and four are complete, all precedence relations are satisfied and the remaining jobs are executed in round robin fashion. First jobs five, six and seven are dispatched to the free processors. They finish at time five. That leaves three free processors and two ready jobs, so jobs eight and nine are started. Job eight finishes at time eight, but job nine finishes at time thirteen. It turns out that for this schedule, adding faster processors that reduce the computation time actually causes the overall schedule to finish later.

Suppose the now frustrated engineer decides that the way to make this system faster is to relax the precedence constraints. Suppose he figures out a way to make jobs seven and eight independent of job four, as shown in figure 29.

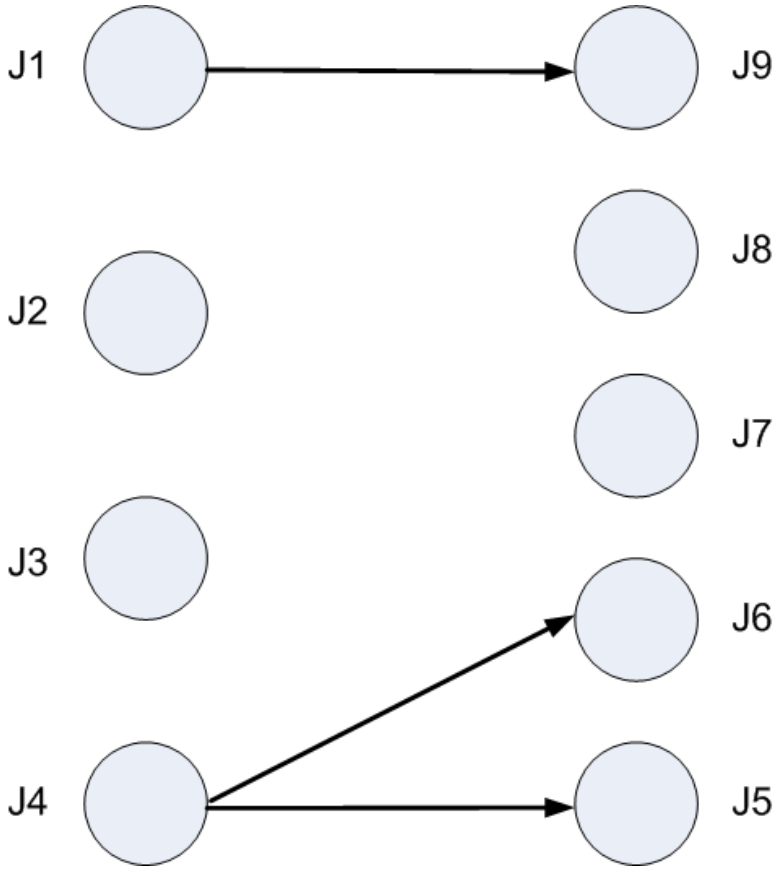

Figure 29: A precedence relationship used to illustrate a Richard's anomaly where an relaxation of precedence constraints results in an increase in schedule length.

The resulting schedule is shown in figure 30. You will probably not be surprised by now to see that relaxing the precedence constraints produces a schedule that takes longer to complete than one with more constraints.

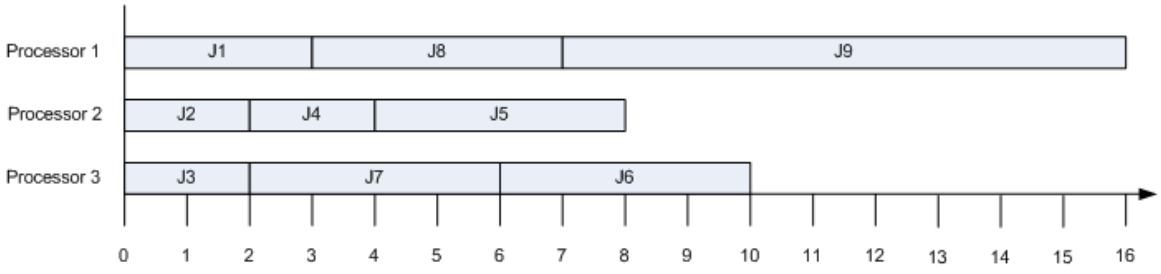

Figure 30: An optimal three schedule for the job set shown in figure 29. In this case, the precedence constraints have been relaxed. The result is that it takes longer to execute with fewer constraints.

Conclusion

The key observation to make as a result of this relatively long story is that real-time systems simply are not like other computer systems. These systems operate in environments where performance is extremely critical. In some cases, a system that does not perform up to

requirements can cause absolutely catastrophic failures. Reports of failing real-time systems can make for spectacular news. You may recall Patriot missile batteries that failed to fire or missed their targets because of failing real-time systems. You may recall stories of interplanetary spacecraft that wandered out into the void or crashed onto other planets because of failures.

Anomalies can arise in real-time systems in ways that amaze many people. Consider what might happen if a real-time system were running the CERN beam dump interlock system and someone decided to upgrade the computer running the Orbit Feedback Controller with a faster model of the existing computer. Suppose that this uncovered a Richard's anomaly that caused the system to be late detecting an out of control beam. The result might be a five hundred megajoule problem. Five hundred megajoules, by the way, is roughly equivalent to the energy in a five-hundred ton naval minesweeper ship moving at 100 miles per hour. This amount of energy can make for a really impressive system "crash."

Figure 31: HMS Ledbury, a British Royal Navy minesweeper ship.

The things to worry about at CERN are not microscopic black holes eating the earth, but more along the lines of the equivalent of a 500 ton minesweeper headed off toward Geneva at 100 miles per hour because of a Richard's anomaly in a real-time system. Even systems as small as the iPod can be affected by anomalies. It is possible to do something as innocuous as adding a faster processor to the system and break it. Real-time systems must be tested in all of the anticipated configurations and supported and tested configurations must be listed. Everyone in the development team should be aware of the issues related to real-time in order to prevent unfortunate results. Even customers must be educated to understand that running an existing software system on better, faster hardware may cause failures.

Although catastrophic failures that make the newspapers will be the most visible outcome of poor choices in real-time system design or testing, there are many less obvious poor results. For example, overspecifying a system can lead to suboptimal design if, for example, a hard-real time system is specified unnecessarily and results in increased direct or indirect costs. Implementing critical functionality in hardware can reduce demands on software and computing systems and possibly make for a less expensive or comlex system as well.

Real-time system design is a complex endeavor and the more knowledge of the underlying problems and solutions is held at all levels, and the better the communication between the people developing the system, the better the result will be.

References

[1] Stankovic J., et al., Deadline scheduling for real-time systems: EDF and related algorithms, Kluwer, Norwell (1998)

[2] nVIDEA, GeForce 256, < http://www.nvidia.com/page/geforce256.html>, (2010)

[3] Inside the Apple iPod Design Triumph, Electronics Design Chain, < http://www.designchain.com/coverstory.asp?issue=summer02>, (2002)

[4] iPod, Wikipedia, < http://en.wikipedia.org/wiki/IPod>, (2010)

[5] Inside Look at the Birth of the iPod, Wired Magazine, < http://www.wired.com/gadgets/mac/news/2004/07/64286>, (2004)

[6] RTXC 3.2 RTOS, Quadros Systems Inc., <http://www.quadros.com/products/operating-systems/rtxc-3.2-rtos>, (2010)

[7] PP5002 Digital Media Management System on a chip, PortalPlayer, <http://web.tagus.ist.utl.pt/~rui.neves/se2005/basedoIpod_5002_brief_0108_Public.pdf>, (2004)

[8] The LHC Control System, European Center for Nuclear Research (CERN), <http://accelconf.web.cern.ch/accelconf/ica05/proceedings/pdf/I1_001.pdf>, (2005)

[9] Facts and Figures, European Center for Nuclear Research (CERN), http://public.web.cern.ch/public/en/lhc/Facts-en.html, (2008)

[10] http://atlas.ch/photos/atlas_photos/selected-photos/detector-site/surface/LHC-PHO-1998-385.jpg

[11] Buttazzo, G., Hard Real-Time Computing Systems, Springer, New York (2005)

[12] Winklmeier, F., The ATLAS High Level Trigger Infrastructure, Performance and Future Developments, IEEE-NPSS (2009)

[13] Yodaiken, V., The RTLinux Manifesto, <http://www.yodaiken.com papers/rtlmanifesto.pdf>

[14] Garland, J. and Anthony, R., Large-Scale Software Architecture, Wiley, Hoboken (2003)

[15] Alexandrescu, A., Modern C++ Design, Addison-Wesley, Boston (2001)