IIT BOMBAY(**ATMEGA 128**)

**Task Scheduling**

Initially (when the payload was a thermopile experiment) there was going to be a lot of data handling. There would have to be error checks on this data and also calculations on how to send it. This seemed like a processor intensive task. However now the experiment has been simplified (TEC measurements), so our team felt that an Operating System would not be needed. Moreover, there is no need for either *preemption or context-switching*. The various tasks that we now have can be handled by a scheduler created from scratch without the overhead of running an OS All the tasks on our satellite will be **deterministic** (i.e. they will have a predefined maximum running time), which makes creating a scheduler a bit easier. Tasks will have different frequencies depending upon their importance. There will be a specific time allocated for a task to finish (since it is deterministic) after which the next task will be run. Since we will be dealing with limited amount of well-defined tasks, a complicated scheduler is not necessary. One which is more robust is preferred. A simple scheduler will be used to handle the various tasks running on the satellite. The scheduler makes it possible to execute various tasks at different frequencies. The scheduler has a list of tasks along with their execution times, and depending upon this list it decides when to execute and how microcontroller time to allocate to each task.

**Basic scheduler design**

We plan to implement a cyclic scheduler. All the tasks will be stored in a circular queue. Each task will be tagged with its maximum execution time as well as a flag. Tasks that have a higher priority will have more instances in that queue. The scheduling function will iterate over the queue. It will execute every task that has its flag set. It will also allocate a time slice equal to the maximum execution time of that task.

**Work done on the scheduler**

Initially we tried to create a scheduler from scratch and program it on to an ATMEGA16. This effort has met with limited success. So an **Operating System (SeOS)** was programmed onto an ATMEGA16. Then it was stripped down by removing all the unnecessary code leaving only the scheduler and a few basic functions. We ran this bare OS with multiple tasks having different priorities. Then a basic scheduler was first implemented in C++ and then ported to an ATMEGA 16 microprocessor.

#include <salvo.h>

typedef unsigned char t\_boolean;

typedef unsigned char t\_temp;

/\* Local flag. \*/

t\_boolean warm = FALSE;

/\* Seat temperature functions. \*/

extern t\_temp UserTemp( void );

extern t\_temp SeatTemp( void );

extern t\_boolean CtrlTemp( t\_temp user, seat );

/\* Moderate-priority (i.e. 8) task (i.e. #1) \*/

/\* to maintain seat temperature. CtrlTemp() \*/

/\* returns TRUE only if the seat is at the \*/

/\* the desired (user) temperature. \*/

void TaskControl( void )

{

while (1) {

warm = CtrlTemp(UserTemp(), SeatTemp());

OS\_Yield();

}

}

/\* High-priority (i.e. 3) task (i.e. #2) to \*/

/\* generate pulses. System ticks are 10ms. \*/

void TaskStatus( void )

{

/\* initialize pulse output (low). \*/

TX\_PORT &= ~0x01;

while (1) {

OS\_Delay(100);

TX\_PORT |= 0x01;

OS\_Delay(5); //DURING THIS PROGRAM RUNS THE LOW PROIRITY TASK RATHER THAT WASTING RESOURCES IN LOOPING

TX\_PORT &= ~0x01;

if (warm) {

OS\_Delay(5);

TX\_PORT |= 0x01;

OS\_Delay(5);

TX\_PORT &= ~0x01;

}

}

}

/\* Initialize Salvo, create and assign \*/

/\* priorities to the tasks, and begin \*/

/\* multitasking. \*/

int main( void )

{

OSInit();

OSCreateTask(TaskControl, OSTCBP(1), 8);

OSCreateTask(TaskStatus, OSTCBP(2), 3);

while (1) {

OSSched();

}  
}

The microcontroller is integrated into the seat, and requires just **four wires for communication** with the rest of the car's electronics – **power, ground, Rx** (to receive the desired seat temperature from a control mounted elsewhere) and **Tx** (to indicate status). The desired temperature is maintained via **TaskControl**(). **Task**-**Status**() sends, every second, either a single 50ms pulse to indicate that the seat has not yet warmed up, or two consecutive 50ms pulses to indicate that the seat is at the desired temperature.

1. *Explicitly manage task switching - cooperative multitasking*

*2. Task switching can only occur at the task level, i.e. directly inside your tasks, and not from within a function called by your task, or elsewhere. This is due to the absence of task stacks and the concomitant ability of the RTOS to save task & state information on the stack. This may have a small impact on the structure of your program*

RTOS FUNDAS

Foreground / background system.. **interrupt system**

Task's *priority* (It may be fixed or variable, unique or shared with other tasks)

Context switch/ task switch - registers

***Preemption***occurs when a task is interrupted and another task is made ready to run. An alternative to a preemptive system is a ***cooperative***system, in which a task must voluntarily relinquish control of the processor before another task may run. It is up to the programmer to structure the task so that this occurs. If a running task fails to cooperate, then no other tasks will execute, and the application will fail to work properly.

Within a [preemptive multitasking](http://en.wikipedia.org/wiki/Preemptive_multitasking) operating system, the scheduler allows every task to run for some certain amount of time, called its [*time slice*](http://en.wikipedia.org/wiki/Time_slice).

If a process does not voluntarily yield the CPU (for example, by performing an [I/O](http://en.wikipedia.org/wiki/Input/Output) operation), a timer interrupt fires, and the operating system schedules another process for execution instead. This ensures that the CPU cannot be monopolized by any one processor-intensive application.

This means that if the CPU requests data from a disk, for example, it does not need to [busy-wait](http://en.wikipedia.org/wiki/Busy-wait) until the read is over, it can issue the request and continue with some other execution; when the read is over, the CPU can be ***interrupted*** and presented with the read

A kernel generally ensures that the highest-priority eligible task is the task that's running

(preemptive scheduling) or will run next (cooperative scheduling).

**Kernel**

switching of tasks (scheduling)

inter task communication *Semaphores*, *messages*, *message queues* and *event flags*

*delay*

event – keyboard mouse

timeout - special handling is invoked

task state - *ready / eligible*, *running*, *delayed***,** *waiting*,*stopped* and *destroyed / uninitialized*.

*Timer*

*Idling - no tasks are running…*

*OS – kernel,timer,services to handle tasks and events..*

*Reentrancy corrupting internal data of a function..*

*Disable interrupts , local vars(vars are to be kept on a fn stack)*

**TERMINOLOGY**

**EVENT DRIVEN PROGRAMMING** - In [computer programming](http://en.wikipedia.org/wiki/Computer_programming), **event-driven programming** or **event-based programming** is a [programming paradigm](http://en.wikipedia.org/wiki/Programming_paradigm) in which the [flow of the program](http://en.wikipedia.org/wiki/Program_flow) is determined by [events](http://en.wikipedia.org/wiki/Event_(computing))—i.e., [sensor](http://en.wikipedia.org/wiki/Sensor) outputs or user actions ([mouse](http://en.wikipedia.org/wiki/Computer_mouse) clicks, key presses) or [messages](http://en.wikipedia.org/wiki/Message_passing) from other programs or [threads](http://en.wikipedia.org/wiki/Thread_(computer_science)).

Event-driven programming can also be defined as an application architecture technique in which the application has a [main loop](http://en.wikipedia.org/wiki/Main_loop) which is clearly divided down to two sections: the first is event selection (or event detection), and the second is event handling. In embedded systems the same may be achieved using [interrupts](http://en.wikipedia.org/wiki/Interrupt) instead of a constantly running main loop; in that case the former portion of the architecture resides completely in hardware.

In computer programming, an **event handler** is an asynchronous [callback](http://en.wikipedia.org/wiki/Callback_(computer_science)) subroutine that handles inputs received in a program. Each *event* is a piece of application-level information from the underlying [framework](http://en.wikipedia.org/wiki/Framework), typically the [GUI toolkit](http://en.wikipedia.org/wiki/GUI_toolkit). GUI events include [key](http://en.wikipedia.org/wiki/Computer_keyboard) presses, [mouse](http://en.wikipedia.org/wiki/Computer_mouse) movement, action selections, and [timers](http://en.wikipedia.org/wiki/Timer) expiring. On a lower level, events can represent availability of new data for reading a file or network stream. Event handlers are a central concept in [event-driven programming](http://en.wikipedia.org/wiki/Event-driven_programming).

The events are created by the framework based on *interpreting lower-level inputs*, which may be *lower-level events* themselves. For example, mouse movements and clicks are interpreted as menu selections. The events initially originate from actions on the operating system level, such as [interrupts](http://en.wikipedia.org/wiki/Interrupts) generated by hardware devices, software interrupt instructions, or state changes in [polling](http://en.wikipedia.org/wiki/Polling_(computer_science)). On this level, [interrupt handlers](http://en.wikipedia.org/wiki/Interrupt_handler) and [signal handlers](http://en.wikipedia.org/wiki/Signal_handler) correspond to event handlers.

Created events are first processed by an *event dispatcher* within the framework. It typically manages the associations between events and event handlers, and may queue event handlers or events for later processing. Event dispatchers may call event handlers directly, or wait for events to be dequeued with information about the handler to be executed

**Contrast with batch programming**

In [batch programming](http://en.wikipedia.org/wiki/Batch_programming), the flow is determined by the programmer. Although **batch programming** is the style taught in beginner programming classes, the more complex event-driven programming is the standard architecture of modern interactive programs.[*[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)*]

Here are two [pseudocode](http://en.wikipedia.org/wiki/Pseudocode) versions of a trivial program to add two numbers:

**Batch version**

read a number (from the keyboard) and store it in variable A[0]

read a number (from the keyboard) and store it in variable A[1]

print A[0]+A[1]

**Event-driven version**

set counter K to 0

repeat

{

if a number has been entered (from the keyboard)

{

store in A[K] and increment K

if K equals 2 print A[0]+A[1] and reset K to 0

}

}

*In this way we can handle any other event instead of a number entered from a keyboard. Then for each such event we can have the appropriate code which is known as* ***EVENT HANDLER(***Because the code for checking for events and the [main loop](http://en.wikipedia.org/wiki/Main_loop) does not depend on the application, many programming frameworks take care of their implementation and expect the user to provide only the code for the event handlers***)****.*

*CREATING EVENT HANDLERS*

1. *Writing subroutines or methods that contain the code of response of a particular event.*
2. *Connect the event handler to the respective event.*
3. *To create the main loop that checks for the event taken place and accordingly calls the corresponding event handler*

***MULTITASKING***

In computing, **multitasking** is a method by which multiple tasks, also known as [processes](http://en.wikipedia.org/wiki/Computer_process), share common processing resources such as a [CPU](http://en.wikipedia.org/wiki/Central_processing_unit). In the case of a computer with a single CPU, only one task is said to be *running* at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by [scheduling](http://en.wikipedia.org/wiki/Scheduling_(computing)) which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a [context switch](http://en.wikipedia.org/wiki/Context_switch). When context switches occur frequently enough the illusion of [parallelism](http://en.wikipedia.org/wiki/Parallel_computing) is achieved

* In [*multiprogramming*](http://en.wikipedia.org/wiki/Multiprogramming) systems, the running task keeps running until it performs an operation that requires waiting for an external event (e.g. reading from a tape) or until the computer's scheduler forcibly swaps the running task out of the CPU. Multiprogramming systems are designed to maximize CPU usage.
* In [*time-sharing*](http://en.wikipedia.org/wiki/Time-sharing) systems, the running task is required to relinquish the CPU, either voluntarily or by an external event such as a [hardware interrupt](http://en.wikipedia.org/wiki/Hardware_interrupt)
* In [*real-time*](http://en.wikipedia.org/wiki/Real-time_computing) systems, some waiting tasks are guaranteed to be given the CPU when an external event occurs. Real time systems are designed to control mechanical devices such as industrial robots, which require timely processing.

Multiprogramming

In the early days of computing, [CPU time](http://en.wikipedia.org/wiki/CPU_time) was expensive, and [peripherals](http://en.wikipedia.org/wiki/Peripheral) were very slow. When the computer ran a program that needed access to a peripheral, the CPU would have to stop executing program instructions while the peripheral processed the data. This was deemed very inefficient.

The first efforts to create multiprogramming systems took place in the 1960s. Several different programs in batch were loaded in the computer memory, and the first one began to run. When the first program reached an instruction waiting for a peripheral, the context of this program was stored away, and the second program in memory was given a chance to run. The process continued until all programs finished running.

Multiprogramming doesn't give any guarantee that a program will run in a timely manner. Indeed, the very first program may very well run for hours without needing access to a peripheral

Cooperative multitasking/time-sharing

When computer usage evolved from batch mode to interactive mode, multiprogramming was no longer a suitable approach. Each user wanted to see his program running as if it was the only program in the computer. The use of time sharing made this possible, with the qualification that the computer would not seem as fast to any one user as it really would be if it were running only that user's program.

Early multitasking systems consisted of suites of related applications that voluntarily ceded time to each other.

**Because a cooperatively multitasked system relies on each process regularly giving up time to other processes on the system, one poorly designed program can consume all of the CPU time for itself or cause the whole system to** [**hang**](http://en.wikipedia.org/wiki/Hang_(computing))**.**

**PREEMPTIVE MULTITASKING**

Preemptive multitasking allows the computer system to more reliably guarantee each process a regular "slice" of operating time. It also allows the system to rapidly deal with important external events like incoming data, which might require the immediate attention of one or another process.

Operating systems were developed to take advantage of these hardware capabilities and run multiple processes preemptively

At any specific time, processes can be grouped into two categories: those that are waiting for input or output (called "[I/O bound](http://en.wikipedia.org/wiki/I/O_bound)"), and those that are fully utilizing the CPU ("[CPU bound](http://en.wikipedia.org/wiki/CPU_bound)"). In primitive systems, the software would often "[poll](http://en.wikipedia.org/wiki/Polling_(computer_science))", or "[busywait](http://en.wikipedia.org/wiki/Busy_waiting" \o "Busy waiting)" while waiting for requested input (such as disk, keyboard or network input). During this time, the system was not performing useful work. With the advent of interrupts and preemptive multitasking, I/O bound processes could be "blocked", or put on hold, pending the arrival of the necessary data, allowing other processes to utilize the CPU. As the arrival of the requested data would generate an interrupt, blocked processes could be guaranteed a timely return to execution.

**Real time**

Another reason for multitasking was in the design of [real-time computing](http://en.wikipedia.org/wiki/Real-time_computing) systems, where there are a number of possibly unrelated external activities needed to be controlled by a single processor system. In such systems a hierarchical interrupt system was coupled with process prioritization to ensure that key activities were given a greater share of available process time.

**Multithreading**

As multitasking greatly improved the throughput of computers, programmers started to implement applications as sets of cooperating processes (e.g. one process gathering input data, one process processing input data, one process writing out results on disk). This, however, required some tools to allow processes to efficiently exchange data.

**CONTEXT SWITCH**

**A context switch is the** [**computing**](http://en.wikipedia.org/wiki/Computing) **process of storing and restoring the** [**state**](http://en.wikipedia.org/wiki/State_(computer_science)) **(**[**context**](http://en.wikipedia.org/wiki/Context_(computing))**) of a** [**CPU**](http://en.wikipedia.org/wiki/Central_processing_unit) **so that execution can be resumed from the same point at a later time. This enables multiple** [**processes**](http://en.wikipedia.org/wiki/Process_(computing)) **to share a single CPU resource. The context switch is an essential feature of a** [**multitasking**](http://en.wikipedia.org/wiki/Computer_multitasking)[**operating system**](http://en.wikipedia.org/wiki/Operating_system)

**A context switch can mean a** [**register**](http://en.wikipedia.org/wiki/Processor_register) **context switch, a task context switch, a** [**thread**](http://en.wikipedia.org/wiki/Thread_(computer_science)) **context switch, or a process context switch. What constitutes the context is determined by the processor and the operating system**

**When to switch?**

There are three scenarios where a context switch needs to occur. They are:

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Context_switch&action=edit&section=2)**] Multitasking**

Most commonly, within some [scheduling](http://en.wikipedia.org/wiki/Scheduling_(computing)) scheme, one process needs to be switched out of the CPU so another process can run. Within a [preemptive multitasking](http://en.wikipedia.org/wiki/Preemptive_multitasking) operating system, the scheduler allows every task to run for some certain amount of time, called its [*time slice*](http://en.wikipedia.org/wiki/Time_slice).

If a process does not voluntarily yield the CPU (for example, by performing an [I/O](http://en.wikipedia.org/wiki/Input/Output) operation), a timer interrupt fires, and the operating system schedules another process for execution instead. This ensures that the CPU cannot be monopolized by any one processor-intensive application.

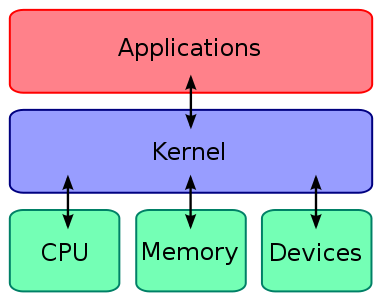
**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Context_switch&action=edit&section=3)**] Interrupt handling**

Some architectures (like the [Intel](http://en.wikipedia.org/wiki/Intel) [x86](http://en.wikipedia.org/wiki/X86) architecture) are [interrupt](http://en.wikipedia.org/wiki/Interrupt) driven. This means that if the CPU requests data from a disk, for example, it does not need to [busy-wait](http://en.wikipedia.org/wiki/Busy-wait) until the read is over, it can issue the request and continue with some other execution; when the read is over, the CPU can be *interrupted* and presented with the read. For interrupts, a program called an [*interrupt handler*](http://en.wikipedia.org/wiki/Interrupt_handler) is installed, and it is the interrupt handler that handles the interrupt from the disk.

The kernel services the interrupts in the context of the interrupted process even though it may not have caused the interrupt. The interrupted process may have been executing in user mode or in kernel mode. The kernel saves enough information so that it can later resume execution of the interrupted process and services the interrupt in kernel mode. The kernel does not spawn or schedule a special process to handle interrupts.

**Round-robin** (RR) is one of the simplest [scheduling algorithms](http://en.wikipedia.org/wiki/Scheduling_algorithm) for [processes](http://en.wikipedia.org/wiki/Computer_process) in an [operating system](http://en.wikipedia.org/wiki/Operating_system), which assigns [time slices](http://en.wikipedia.org/wiki/Preemption_(computing)#Time_slice) to each process in equal portions and in circular order, handling all processes without [priority](http://en.wikipedia.org/wiki/Priority). Round-robin scheduling is both simple and easy to implement, and [starvation](http://en.wikipedia.org/wiki/Resource_starvation)-free

**Kernel**

[](http://upload.wikimedia.org/wikipedia/commons/8/8f/Kernel_Layout.svg)

it is a bridge between applications and the actual data processing done at the hardware level. The kernel's responsibilities include managing the system's resources (the communication between [hardware](http://en.wikipedia.org/wiki/Computer_hardware) and [software](http://en.wikipedia.org/wiki/Computer_software) components).[[1]](http://en.wikipedia.org/wiki/Kernel_(computing)#cite_note-Wulf74-0) Usually as a basic component of an operating system, a kernel can provide the lowest-level [abstraction layer](http://en.wikipedia.org/wiki/Abstraction_layer) for the resources (especially [processors](http://en.wikipedia.org/wiki/Central_processing_unit) and [I/O devices](http://en.wikipedia.org/wiki/Input/output)) that application software must control to perform its function. It typically makes these facilities available to [application](http://en.wikipedia.org/wiki/Application_software) [processes](http://en.wikipedia.org/wiki/Process_(computing)) through [inter-process communication](http://en.wikipedia.org/wiki/Inter-process_communication) mechanisms and [system calls](http://en.wikipedia.org/wiki/System_call).

After this, the kernel does not typically execute directly, only in response to external events (*e.g.*, via system calls used by applications to request services from the kernel, or via [interrupts](http://en.wikipedia.org/wiki/Interrupt) used by the hardware to notify the kernel of events). Additionally, the kernel typically provides a loop that is executed whenever no processes are available to run; this is often called the *idle process*.

* The [Central Processing Unit](http://en.wikipedia.org/wiki/Central_Processing_Unit) (CPU, the processor). This is the most central part of a computer system, responsible for *running* or *executing* programs on it. The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors (each of which can usually run only one program at a time)
* The computer's [memory](http://en.wikipedia.org/wiki/Random-access_memory). Memory is used to store both program instructions and data. Typically, both need to be present in memory in order for a program to execute. Often multiple programs will want access to memory, frequently demanding more memory than the computer has available. The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough is available.
* Any [Input/Output (I/O)](http://en.wikipedia.org/wiki/Input/output) devices present in the computer, such as keyboard, mouse, disk drives, printers, displays, etc. *The kernel allocates requests from applications to perform I/O to an appropriate device (or subsection of a device, in the case of files on a disk or windows on a display) and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device).*

Kernels also usually provide methods for [synchronization](http://en.wikipedia.org/wiki/Synchronization_(computer_science)) and [communication](http://en.wikipedia.org/wiki/Inter-process_communication) between processes (called *inter-process communication* or IPC)

In a [pre-emptive multitasking](http://en.wikipedia.org/wiki/Pre-emptive_multitasking) system, the kernel will give every program a slice of time and switch from process to process so quickly that it will appear to the user as if these processes were being executed simultaneously. The kernel uses [scheduling algorithms](http://en.wikipedia.org/wiki/Scheduling_algorithm) to determine which process is running next and how much time it will be given. The algorithm chosen may allow for some processes to have higher priority than others. The kernel generally also provides these processes a way to communicate; this is known as [inter-process communication](http://en.wikipedia.org/wiki/Inter-process_communication) (IPC) and the main approaches are [shared memory](http://en.wikipedia.org/wiki/Shared_memory), [message passing](http://en.wikipedia.org/wiki/Message_passing) and [remote procedure calls](http://en.wikipedia.org/wiki/Remote_procedure_call) (see [concurrent computing](http://en.wikipedia.org/wiki/Concurrent_computing)).

Other systems (particularly on smaller, less powerful computers) may provide [co-operative multitasking](http://en.wikipedia.org/wiki/Co-operative_multitasking), where each process is allowed to run uninterrupted until it makes a special request that tells the kernel it may switch to another process. Such requests are known as "yielding", and typically occur in response to requests for interprocess communication, or for waiting for an event to occur.

Device management

* Using a software-simulated [interrupt](http://en.wikipedia.org/wiki/Interrupt). This method is available on most hardware, and is therefore very common.
* Using a [call gate](http://en.wikipedia.org/wiki/Call_gate). A call gate is a special address stored by the kernel in a list in kernel memory at a location known to the processor. When the processor detects a call to that address, it instead redirects to the target location without causing an access violation. Requires hardware support, but the hardware for it is quite common.
* Using a special system call instruction. This technique requires special hardware support, which common architectures (notably, [x86](http://en.wikipedia.org/wiki/X86)) may lack. System call instructions have been added to recent models of x86 processors, however, and some (but not all) operating systems for PCs make use of them when available.
* Using a memory-based queue. An application that makes large numbers of requests but does not need to wait for the result of each may add details of requests to an area of memory that the kernel periodically scans to find requests.