Part I Revision

As discussed in the first part of the module, real-time systems are classified into three categories: **hard real-time** (critical that system responds correctly on time), **soft real-time** (can occasionally miss deadline), **firm real-time** (soft real-time but no benefit when delayed). A single system may have a combination of these classes of real-time systems.
A **time-aware system** makes explicit reference to time, while a **reactive system** must produce output within deadline, which may contain constraints on input and output time variability, known as **input jitter** and **output jitter**. Control systems are reactive systems.
Characteristics of real-time systems: guaranteed response times, concurrent control of separate system components, facilities to interact with special purpose hardware, support for numerical computation, them being large and complex, and extreme reliability and safety.
Ada is chosen as the language to program real-time systems in this module, as it provides facilities to support real-time programming, and has a **Ravenscar Profile** to support a small certifiable real-time system for high-integrity systems, as well as analysability.

Ada Programs

An Ada program can have one or more **library units**, each of which can either be a **subprogram** (a procedure or function), or a **package** (used for encapsulation and modularity, with a specification and a body). Each unit can be separately compiled with the compiler, allowing a package specification to be compiled without its body.
Blocks: as a block-structured language, each Ada block contains *declare* (definitions of types, objects, subprograms, etc.), a sequence of statements after *begin*, followed by any exception handlers required in *exception*, and finally terminates in *end*. A block can substitute the place of a statement anywhere in an Ada program.
Attributes: variables can have *VarName'AttriName* to retrieve variable properties, such as *'Range*, *'First* and *'Last*.
Limited types: when a type is declared *limited*, objects of the type cannot be assigned values of the same type. This prevents re-referencing or 'shadow copying' of a variable to another.
Object-oriented programming (OOP) in Ada: Ada supports OOP through tagged types and class-wide programming and interfaces, but it does not require programs to be written in an OOP style.
Tagged types: tagged types preserve a common interface across differing implementations, useful in maintaining structural invariance in type derivations.

The Ravenscar Profile

The **Ravenscar Profile** is a subset of Ada supporting the development of analysable real-time systems. It places more restrictions on the programs within the profile to maintain the analysability of the system.
For Ada primitive mechanisms, the included components are **tasks** and **protected** objects, along with clocks, time and timing events. For real-time abstractions, the profile allows **periodic** and **sporadic tasks**, with shared data. Two schedulable systems are provided: priority-based scheduling and earliest deadline first (EDF) scheduling (see Part I notes) within each priority level's queue. (N.B. EDF across priority levels not supported.) Low-level programming is also considered, including interrupt handling, but strictly not part of the official profile.

Tasks

In Ada, a **task** is the unit of concurrency. Each task must be explicitly declared at any program level, but they are always created implicitly at runtime, when execution of the program enters the scope of a task declaration, or when the task is dynamically allocated by an allocator.
Tasks may communicate and synchronise via **rendezvous** (a form of synchronised message passing), shared variables, or protected objects.
A **task type** consists of a **specification** (name, optional discriminant — in/out variables, visible and invisible parts for entries), and a **body**, which may contain an operation loop or a simple set of statements, depending on the purpose of the task.
If a task is wrapped in a procedure, when the procedure starts executing, the task always start before the body of the procedure, and prevents the procedure from terminating until all tasks

initialised by the procedure terminates. Wrapping multiple tasks in a procedure can ensure their simultaneous initialisation.
Tasks can be **dynamically created** by using the *new* operator on an **access type** to the task type. It can also be created by giving non-static values to the bounds on an array of tasks.
Phases of the task execution: **activation** (elaborating the declaratives, creating and initialising local variables), **normal execution** (executing statements from the task body), and **finalisation** (executing finalisation code associated with objects in declaratives).
Ravenscar restrictions on tasks: no task hierarchies (removes parent/child task synchronisation), no dynamic task creation, and no task entries (for analysability, therefore no task rendezvous).

Shared Variables
Without the use of more advanced primitives such as rendezvous or protected objects, several tasks may be able to safely pass data through simple **shared variables** if they are synchronised in some other way, with the examples of tasks A and B both accessing the same variable V:
- A accesses V before activating B;
- A accesses V on activation, and B is waiting for A to activate before accessing V;
- A accesses V, and B is waiting for A to terminate, and then accesses V;
- A accesses V, then blocks on an entry call, during which B accesses V;
- A blocks on an entry call in B, during which B accesses V, then after the entry call A accesses V.

Otherwise, the values read from or written to shared variables may be semantically invalid, due to compiler optimisation of memory write-backs. Synchronisation ensures the consistency of data, as well as the visibility of changes in data.
The compiler can also be prevented from optimising memory write-backs that break the ordering of access through the use of **volatile pragma**, which can be applied to object declarations, non-inherited component declarations, and full type declarations.
It is also possible to force a set of consecutive actions to be indivisible (hence producing a consistent state) with the use of **atomic pragma**, which should be as small as possible as large atomic actions significantly reduce the performance of the program.

Protected Objects
Protected objects allow the encapsulation of data and operations, and making data updates atomic. Barriers can also be placed on operations for conditional synchronisation. There are three types of protected operations: procedures (read/write data atomically one at a time), entries (rendezvous between tasks, potentially with barrier), and functions (read only, allows concurrent execution). Procedure and function calls are mutually exclusive, so are calls to a protected entry.
Each **protected type** only has subprogram and entry declarations in the public part, with possible object declarations in the private part in addition to subprogram and entry declarations.
Private entries: protected types may have private entries that can be used during requeue operations, but otherwise not visible to external tasks or program bodies.
Language restrictions: potentially blocking operations such as entry calls and delays cannot be included in a protected object, but calls to an external protected procedure or function are exempt.
Ravenscar restrictions: no nested protected objects, no dynamic protected object creation (*new*).

Conditional Synchronisation
With the use of barriers (guards) on an entry, the calling task is suspended (blocked) if the barrier evaluates to false, or if another task is active in the protected unit. Barriers are evaluated after each entry call and when any task leaves any protected procedure or entry.
Function returns do not require barrier re-evaluation as they cannot possibly have changed the data due to read only access.
N.B. parameters passed into the entry cannot be accessed by the guard until the entry is accepted past the barrier, as their values may have changed while the calling task was blocked — not a safe shared variable. This limits the expressive power of entry calls, which will be resolved with *requeue* later. i.e. it is not possible to compare a protected object variable with an entry parameter at guard.
Count attribute: defines the number of tasks queued on an entry.
The proxy model: implemented in Ada. When a task is accepted past the barrier and finishes executing the entry body, it checks whether any task blocked by barriers can now be accepted. If

there are any, it wakes each calling task and services the entry on the calling task's behalf. It only exits the protected object when there are no more entry calls waiting on acceptable barriers.
Ravenscar restriction: only one entry in each protected type, and only one task can be queued on that — consequently no 'Count in barrier. As otherwise the proxy model will result in more than one task's entry call being executed, reducing analysability. Nested and private entries are disallowed. The entry barrier can only be a single boolean variable or literal, so no comparisons allowed.

Avoidance Synchronisation
Unlike conditional synchronisation, in **avoidance** synchronisation, a task is not accepted by an entry if its request demand cannot be met. The condition for a demand to acceptable is expressed as a guard on the action of acceptance.
Sometimes, it may be necessary to establish an order for accepting tasks called on the entry, which may require measuring the information in the **discriminant** (parameter variables) of the entry call. But this is not possible in Ada due to the limited expressive power.
Therefore, a **double interaction** is necessary for the request, first receiving the required information then determining whether the entry can accept the calling task. However, this cannot be implemented as an atomic action in Ada, resulting in concurrency problems between the two steps (calls). If the entry call is aborted after receiving the information but before the entry can be accepted, an inconsistent internal state will be created. To resolve this problem, Ada uses the *requeue* facility to remove the need of the second call, by moving the first call to another entry's queue directly, or simply moving it back into the queue to be tried again when conditions change.
requeue: a *requeue* statement can move a waiting task that has passed the first step/call of the interaction to another entry for guard evaluation, perhaps into a FIFO queue. *requeue* always completes the current entry call. It can be unabortable or abortable (*with abort*). Abortable *requeue*s reinstate any timeout and allows the queued task to be aborted, vice versa for unabortable *requeue*s. *requeue* can also be used for simply moving tasks to other entries.

Time in Ada
Real-time programming requires interfacing with time (measuring passage of time, delaying tasks, programming timeouts, etc.), as well as representing timing requirements (rates of execution, deadlines). The source of time can either from environmental timeframes (e.g. GPS) or an internal hardware approximation clock.
Calendar: the basic clock package, with range-defined time representations and overloaded operators (as functions) between time representations. The time base is implementation-defined with no requirement to be synchronised with UTC. Duration is a fixed point scalar type, and is implementation dependent.
Real Time Clock: a more precise (millisecond level) and monotonic (ever-increasing) clock. It is the only Ravenscar supported clock. It is not affected by timezone changes.

Delaying Tasks
A task may need to delay its execution, either for a relative period of time, or resume at a certain time in the future. **Busy waiting** is not a good idea, where the task spins in a loop until timing conditions are met, since it occupies computational resources — unless the timing is too small for effective task switching.
Ada provides two delay statements: *delay* and *delay until*. *delay* sets a certain amount of suspension period for the task, while *delay until* (used for both calendar and real time clock) sets an absolute time in the future for the task to resume. Due to overheads, *delay until* is usually more accurate in setting a resumption time.
Delay statements can suffer from **drifts**, which consists of local and cumulative drifts. While local drift is unavoidable due to system overheads, cumulative drift is only present in *delay* but not in properly constructed *delay until*, as the effects of other tasks overrunning will not directly affect the period of delay in the latter case.
Timeouts: accompanied by delay statements, timeouts can be set with **select statements** (select… or…, select… then abort…; to be introduced later). This allows code executing for excessive amounts of time to be aborted early. This also allows imprecise computations to be iteratively improved until the allocated amount of time runs out.

Timing Events

Events are one-off stateless computation abstractions that can be triggered/fired/invoked when required. Interrupts are events. **Timing events** are triggered when a defined clock reaches a specified value, allowing defined code to be executed at a future point.
Each timing event generates an interrupt, for which a handler is needed. The handler can be an access type to a procedure inside a protected object which handles the event processing.

Periodic Tasks

Ada does not directly support **periodic tasks** or other real-time abstractions, but the language does provide the necessary low level primitives to implement these abstractions.
Programming Deadline Monotonic Ordering (see ARTS I): a priority can be assigned to a task with the use of the Priority pragma, which requires pre-determination of the deadline for that task in DMPO.
Programming periodic tasks: use a loop in the body of the task to perform the task computations, then at the end of the loop use *delay until* to set the next release after a fixed interval from the previous release of this task. For an offset start, a *delay until* can be used at the very beginning of the task body to delay the task before starting.

Sporadic and Aperiodic Tasks

Non-periodic tasks with a **minimum inter-arrival time** (**MIT**) is a **sporadic task**; otherwise it is an **aperiodic task**.
Programming aperiodic tasks: an **aperiodic server** (see ARTS I) can be programmed with a protected object, with an interrupt handling procedure that starts processing the interrupt (lost if not handled fast enough), and entry to unblock the calling task when the interrupt is started. The entry can then be called in the task body within an event loop, thus triggering at most once during each available interrupt period.
Programming sporadic tasks: similar to the protected object for the aperiodic server, but contains a MIT variable to define the minimum interval. The entry of the protected object outputs the next arrival time by adding MIT to the clock time at the end of the entry call. Then the task actions are performed in the main task body. Interrupts may need to be turned off during MIT for analysability.

Jitter Control

A task may compete with other running tasks in the system, causing considerable variations of starting and finishing times in each period, known as **input** and **output jitters**. They need to be controlled as operations like reading and writing sensor data have time sensitivity.
Controlling jitter: use a timing event for input and a separate timing event for output. Use a task for processing the input data to produce the output. During the atomicity-protected start of processing, a timer handler is set to read/write data at a certain, fixed time in the future. At that time, an interrupt is generated by the timing event, causing the read/write to be performed, and the next timing event for read/write to be scheduled in the same way. This creates a periodic time handler interrupt, controlling the jitter. Operations other than read and write (e.g. normal computations) happen under normal priority (rather than the high interrupt priority), as to not disrupting DMPO.
A controlling task can be used to perform read (entry passed as soon as data ready), computation (normal priority procedure), and write (entry queued as soon as computation finished, but passed only when output jitter allows) in a loop, ensuring both jitters respected.
However, since all read and write operations now operate at interrupt priority level, this itself may cause problems in scheduling. Therefore the time taken for these operations must be minimised.
Possible mitigations: running sensor and actuator controllers as periodic tasks with representative priority/deadline requirements (multiple task approach); allowing a task to dynamically change priority (single task multiple priority approach). There is flexibility to choose.

Priority-based Systems

Ada supports **priority-based systems**. It is usually preemptive to give good response time to high priority tasks, and has a reasonable range of priorities for good schedulability. Some form of priority inheritance is used to bound blocking. The scheduling scheme can be specified with the Task_Dispatching_Policy pragma. In Ada programming, higher number means higher priority.

As mentioned earlier, each instance of any type of task can have a separate priority, with the use of the Priority pragma with varying priority parameter.

Priority ceiling locking: each protected object is assigned a ceiling priority which is higher than or equal to priorities of all its calling task (immediate priority inheritance, see ARTS 1). The protected object has mutually exclusive access, just like a resource. If the calling task's active priority is greater than the celling, an error is triggered as this situation is erroneous. The proxy model can be used to execute code from computations released by an accepted task, and then resume the original calling task. There will be no context switching away from the original calling task.

Active priorities: a task's priority may change when entering a protected operation (priority celling); on activation (priority inheritance); during rendezvous (inheritance), but no inheritance when waiting for task termination. The active priority changes to reflects this.

Task scheduling/dispatching: The active priority of a task determines its order of dispatching. By default, Ada uses preemptive priority-based scheduling, other policies include *FIFO_Within_Priority*, which is present in the real-time annex; Fixed Priority (programmer-set before active priority changes); and Dynamic Priority. More details in the next section.

FIFO_Within_Priority: a FIFO queue is established for each priority level. When a task becomes runnable it is placed at the back on the run queue for its priority. When it is preempted, it is placed at the front of the queue, to preserve priority celling.

Dynamic priority: tasks are allowed to change their base priorities. A task may have a priority range, if its priority is changed to be outside this range by *Set_Priority*, it will be set to the default priority. Any change should be applied as soon as possible barring some circumstances related to aborting. The active priority of the task will consequently be changed. Example: EDF (see ARTS I).

Dynamic ceiling priority: protected objects are allowed to change their celling priorities by assigning to their Priority attribute. This is only allowed from within the protected object. In addition, change of the ceiling value can only occur at the end of the protected operation to prevent causing problems by lowering the priority ceiling.

Ravenscar restriction: preemptive fixed priority queues only, no EDF or Round-Robin queues.

Configuring Dispatching Policies

All the dispatching/scheduling policies are defined using a pragma and may also be supported by a language-define package: *pragma Task_Dispatching_Policy(Policy_Identifier)*. This pragma is a **configuration pragma**, which means it should appear with the main program.

Non-preemptive scheduling (*Non_Preemptive_FIFO_Within_Priorities*): no preemption when a higher priority task becomes runnable, so a low priority task may block a runnable high priority task from executing. FIFO is used for the ready queue.

Round-Robin with priorities (*Round_Robin_Within_Priorities*): a quantum is assigned to each task, which equals to the execution time budget of a runnable task added to the tail of the ready queue. Preempted tasks are instead moved to the head of the ready queue and retain their budgets. The budget is consumed by task execution time. If the budget runs out and the task does not inherit a higher priority or execute with a protected operation (such as when inside a protected object), it is moved back to the tail of the ready queue, with budget reset. FIFO is used for the ready queue.

Earliest Deadline First (*EDF_Across_Priorities*): requires formal representation of task deadlines, which are used to control scheduling. EDF-compatible data sharing must be available between tasks. A task may delay itself with one deadline and awake with another to manage risk conditions. Ready queues in EDF are **priority queues**, and are EDF-ordered. All EDF tasks are initially released on the highest priority EDF-ordered queue, and they will have their **preemption level** (represented by a numerical priority) set and move between priority queues when they lock protected objects, which requires the consideration of ceilings.

Mixed dispatching policies can be used together in Ada, as long as they apply to non-overlapping ranges of priorities: *pragma Priority_Specific_Dispatching (pid, pri_exp_1, pri_exp_2, …);* and a dispatching policy of *pid* will be set for tasks with priorities between *pri_exp_1* and *pri_exp_2*.

Ravenscar restriction: dispatching with EDF queues not supported.

Device Interface

Device interfaces are used to control device operations and data transfer, usually through a set of registers. Both the device memory and registers can be accessed with separate sets of assembly instructions, known as **port-based I/O**. Another strategy on some devices is to map some addresses to the memory and others to registers, known as **memory-mapped I/O**. This may consist of varying numbers of **control & status registers** (CSR) and **data buffer registers** (DBR), depending on the nature of the device.

Interrupt-driven Systems

In non-safety-critical systems, **interrupts** may be allowed as a control mechanism. Several models exist for interrupt-driven systems, include interrupt-driven program-controlled, interrupt-driven program-initiated (DMA), and interrupt-driven channel-program controlled. The latter two can cause cycle stealing from the processor, making worst-case execution time analysis difficult.
Elements required for interrupt-driven systems: context switching mechanisms (state preservation, state switch, state restoration, hardware support); interrupting device identification and reasoning (vectored mechanism — follows address to interrupt handler checker which redirects if handler exist, status word mechanism, or polling mechanism); interrupt control (status, mask interrupt or level-interrupt mechanisms); priority control (urgency, static or dynamic).
Language requirements: modularity and encapsulation facilities (some implementation details machine-dependent, need to separate non-portable sections of code from portable ones, e.g. Ada packages); an abstract device handling model (model devices as fixed tasks in parallel, with communication and synchronisation/interrupt requirements).
Abstract model requirements: facilities for addressing and manipulating device registers (variable, object, channel, etc.); suitable representation of interrupt (procedure calls, asynchronous notifications, etc.); scope and context switch support for interrupts.

The Interrupt Model

An **interrupt** represents a class of events that are detected by the hardware or systems software, the **occurrence** of which consists of its **generation** (underlying system makes interrupt available to the program) and **delivery** (invocation of the interrupt handler). Between generation and delivery is a pending state, the time spent in which is called latency. The handler is invoked once per delivery.
When an interrupt is being handled, further interrupts from the same source are blocked (either remains pending or is lost). Certain interrupts are reserved (e.g. clock interrupt for *delay* only). Non-default interrupts have defaults handlers. Each interrupt has a unique identifier like address of the interrupt vector.

Interrupt Support in Ada

A device driver as a **hardware task** manipulates device registers and responds to interrupts, which are modelled as **hardware protected procedure calls**. Shared memory device registers are best supported.
Representation aspects: representation is a comprehensive set of facilities for specifying the implementation of data types, which can be **attribute definition clause** (size, alignment, etc. of tasks and addresses, user-defined structure), **enumeration representation clause** (specifies the internal codes for the literals of the enumeration type) and **record representation clause** (the storage representation of records — offsets and lengths of components).
Interrupt handler identification: provided by two pragmas: *pragma Interrupt_Handler(name)* and *pragma Attach_Handler(name, exp)*. The first is in library level only, allows dynamic association with parameterless protected procedures serving as handlers. The second associates a handler with an interrupt identified by the expression, attached when the protected object is created. It can raise *Program_Error*.
Dynamic handler attachment: requires the definition to be changed that the pragma indicates the intention for handler to be used as an interrupt handler, in the public part of protected type declaration.

Reliability and Fault Tolerance

Sources of system failure: inadequate specification, design errors, processor failure, communication interference. Only design errors in software are discussed here.

Safety: preventing hazardous conditions that can cause serious danger to human, the operating equipment or the environment. Systems have an element of risk in their use are unsafe systems.

Reliability: a measure of success with which a system conforms to some authoritative specification of its behaviour.

Dependability: a system should be available, reliable, safe, confidential, integral and maintainable.

Failure: when the behaviour of a system deviates from that which is specified for it, this is called a failure, which is a result of unexpected problems internal to the system. The problems are called errors or **faults**.

Fault types: transient faults (start at particular times, remain for a while then disappear, frequently seen in communication systems); permanent faults (remain until repaired); intermittent faults (occur from time to time, for example overheating); software faults (either easily reproducible or hard to reproduce like race conditions, can remain dormant for long periods like memory leaks).

To achieve reliable systems, **fault prevention** (prevent faults happening) and **fault tolerance** (continue functioning under faults) are used, both approaches attempt to produces systems which have well-defined failure modes.

Fault prevention has two stages: fault avoidance and fault removal. The former attempts to limit the introduction of faults during construction phases by various methods, such as using reliable components and rigorous specifications. The latter finds and removes causes of errors through verification and testing. However, testing can never be exhaustive and remove all potential faults.

Fault tolerance does not require interruption to the access of the system, which is advantageous. It has several levels, including full fault tolerance (limited period of full performance continued operation under fault), graceful degradation (reduced performance during error recovery), and fail safe (temporarily halt in a safe state), depending on the requirement from application.

Redundancy: fault tolerance requires extra elements introduced into the system for fault detection and recovery, known as protective redundancy. The aim is to minimise the redundancy while maximising reliability, subject to cost and size constraints — also given the fact that redundancies themselves increase system complexity and lead to more failure points, therefore they should be separated out from the rest of the system.

Software Fault Tolerance

Software fault tolerance is used for detecting design errors, which is either static (N-version programming) or dynamic (detection and recovery.

Static fault tolerance (N-version): multiple versions of equivalent programs based on the same specification function independently on the same inputs, with no interaction between groups. The results of their executions are compared by a driver process, if there are differences the majority is taken. This requires adequate initial specification, independence of effort in different versions, and adequate budget (sometimes multiple independent versions will be cost-prohibitive).

Dynamic fault tolerance: as an alternative to static fault tolerance, dynamic redundancy has four phases: error detection, damage confinement and assessment (potential further fault spread since detection), error recovery (restore system into a continuable state potentially with degraded functionality), fault treatment and continued service (seek to prevent future faults of the same kind). Details explained as followed.

Error detection: could be environmental detection (hardware or operating system, like illegal instruction or null pointer), or application detection (e.g. replication, timing, coding).

Damage confinement and assessment: two closely related families of techniques. Damage confinement structures the system to minimise damage caused by a faulty component (firewalling). **Modular decomposition** provides static damage confinement, allowing data to flow through well-defined pathways. **Atomic actions** provide dynamic damage confinement, moving the system from one consistent state to another.

Error recovery: the most important phase with two possible approaches: **forward error recovery** (FER), which continues from an erroneous state by making selective corrections to the system state and its environment; and **backward error recovery** (BER), which restores the system to a previous safe state and executes an alternative section of the program providing the same

functionality (analogues with N-version). BER establishes recovery points with checkpointing of system states.
FER and BER comparison: BER has the advantage of clearing the erroneous state and thus not requiring determining the location of the fault, including correcting unanticipated faults that are hard to pinpoint. However, BER cannot correct environmental errors like FER does. In concurrent process interactions, BER also suffers from the domino effect when concurrent tasks with different recovery points fail, causing difficulties in maintaining a constant state in recovering.
Exception handling: a FER mechanism passing control to a handler to initiate the recovery procedures. This allows the system to cope with abnormal environmental conditions, tolerate program design faults and provide a general purpose error detection and recovery facility. Discussed in more detail in the next section.
Fault treatment and continued service: to prevent the recurrence of the error, the fault should be eradicated. Two stages are involved in this process: fault location (with error detection techniques) and system repair (might require live system modification if the system is non-stop).

Exception Handling
Exceptions can be raised from environmental detection and application error detection. Exceptions can either be **synchronous** (immediate results of tasks attempting inappropriate operations) or **asynchronous** (raised some time after the erroneous operation, in the same or a different task). The latter are often also called **asynchronous notifications** or **signals**.
Exception classes: environmental detection raise synchronously (e.g. divide by 0), application error detection raised synchronously (e.g. assertion error), environmental detection raised asynchronously (e.g. application error checkers), application error section raised asynchronously (the failure of one task causing another to fail).
Exception handler domains: multiple handlers can be associated with a particular handler, with triggering differentiated by associated domains, with varying accuracy in identification. In Ada, the domain is normally the triggering program block, or a procedure or function.
Error propagation: if there is no handler associated with the block or procedure, a programmer error can simply be triggered and reported at compile time. As an exception raised in a procedure can only be handled within its calling context, an better strategy is to **propagate** the exception, subject to a scope limit — propagating beyond the scope will trigger a catch-all handler. What happens when an exception is triggered can be categorised into either the **resumption** or the **termination model**.
Termination model: single threaded programs will terminate the exception-triggering procedure call tree on exception, and resume at the end of the exception handler — so do individual threads facing exceptions in a multi-threaded program. Whether parent or sibling threads will have exceptions propagated to them depends on the design.
Resumption model: the exception handler may be able to correct the problem, allowing the invoking process to continue. However, this may be hard to do in real-time systems, as exception is often raised **asynchronously** and cannot help the current task's continued execution.

Ada Support for Exception Handling
Ada supports explicit declaration of exceptions, the termination model, propagation of unhandled exceptions and a limited form of exception parameters. Exceptions can either be declared by a language keyword *: exception*, or with the predefined package *Ada.Exceptions*. Each declared exception has an associated *Exception_Id* to uniquely identify it.
An exception can be raised explicitly to an **occurrence** with *raise … with "…".* Exception handlers can be declared at the end of each block, subprogram or task as a sequence of statements, which can be individually guarded with the conditions in *when … =>*, and caught-all with *others* and *Reraise_Occurrence(Exc).* If no handler is defined within the limit, the exception is propagated to an enclosing block, subprogram, or the called or calling task.
Difficulties in Ada exceptions: exceptions in packages are hard to find, parameter passing is by string only, exceptions can be propagated outside scope of declaration.

Asynchronous Exceptions and Notifications

Asynchronous exceptions are exceptions that are not raised synchronously and not necessarily on a single task. It is raised some time after the operation that caused the error in not necessarily the original task, which means that the current task may not be equipped to handle it. **Asynchronous notifications** are used to mitigate this issue.

Asynchronous notifications is a mechanism whereby one task can gain the attention of another without the latter waiting. It can be used as a basis for error recovery between concurrent systems. It has both resumption (like software interrupt) and termination models. In the termination model, each task specifies a domain of execution during which it can receive an asynchronous event.

Asynchronous notifications are needed to allow quick response to a condition detected by another task, for example error recovery or urgently required mode changes. They can also allow efficient scheduling of partial/imprecise computations, and handling user interrupts. Polling is too slow to be an alternative for these requirements.

In Ada, asynchronous notifications allow the application to handle interrupts and timing events, such as **asynchronous transfer of control** (ATC) requests and **task abortions**.

Asynchronous Transfer of Control

Asynchronous transfer of control allows a task to open up a window to wait and access an entry in a protected object or rendezvous in another task, while also waiting for a limited amount of time or performing another operation. If the waiting or the another operation performed complete before the task can enter the protected entry or rendezvous, then the task aborts the rendezvous and continue execution. For ***select*** *protectedObject.protectedEntry* ***then abort*** *alternativeStmt*, four possibilities exist:

- *protectedEntry* available immediately: task enters *protectedEntry* and does not execute *alternativeStmt*;
- *alternativeStmt* finishes before *protectedObject.protectedEntry* becomes available: task carries on without waiting further for *protectedObject.protectedEntry*;
- *protectedEntry* available during *alternativeStmt*: if *protectedEntry* finishes before *alternativeStmt*, *alternativeStmt* is aborted; otherwise, both are completed.

The ***abort*** statement is intended for when recovery by the errant task is not deemed possible. Any task may abort any other named task, which makes the aborted task as well as its dependent **abnormal**. When a task becomes abnormal, every construct it is executing is aborted immediately unless it is included in an **abort-deferred operation**. If a construct is blocked outside an abort-deferred operation (other than at an entry call), it is immediately completed. Other constructs must complete no later than certain important task actions.

<u>Abort-deferred operations</u>: protected actions include waiting for an entry call to complete after cancellation, waiting for termination of dependent tasks, executing an *Initialize*, *Finalize*, or assignment operation of a controlled object, and certain actions resulting in bounded error.

Abort actions should be used sparingly due to its strong real-time implications.

Timing Faults

Even in a system mathematically proved to make the deadlines, they can still be missed for reasons such as inaccurate worst-case calculations, underestimation of blocking times, or the system working outside its designed parameters. These are known as timing faults.

Miscalculation of a higher priority task’s worst case response time (WCET) could increase its interference on lower priority tasks, causing them to miss deadline in addition to itself.

Therefore, it is necessary to be able to detect deadline misses, WCET overruns, sporadic events occurring more frequently than expects, and resource overuses. Except direct deadline misses, the other three may or may not cause a deadline miss. Appropriate actions must be taken on detection, with either forward or backward error recovery.

In Ada, ATC can be used to detect deadline misses in periodic or sporadic tasks, but sometimes it is better to raise an asynchronous event to allow the deadline-missing task to continue execution, which requires the use of a **watch dog timer**. A watch dog timer is attached to a timing event, and a handling procedure waiting on the control entry begins handling the timing fault if *Call_In* does not arrive at the watch dog at regular timings.

For damage confinement, the consequences of an error should be restricted to a well-defined region of the program. A task causing the timing error may not be the one responsible, especially if it ordinarily has a lot of slack. A lower priority task with less slack may have missed its deadline. Therefore, to determine the responsible task, we must monitor the **execution time** of a task.
Execution Time Clock: in Ada, each task has a notional clock that starts from zero at task creation, called the execution time, of type *CPU_Time*. Execution-time timers allow a protected procedure to be executed when a task's clock reaches a specific values, which can be used to catch overrun tasks. Not available in Ravenscar due to the need to simplify implementation.

Detecting Sporadic Overruns

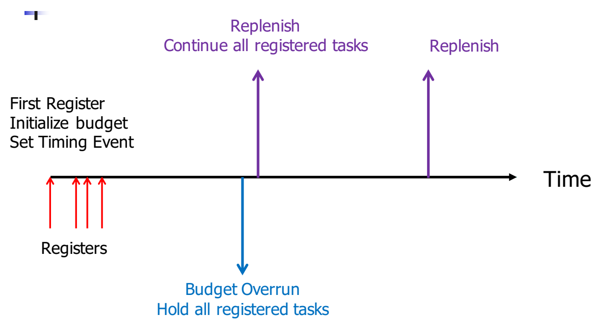
A sporadic event overruns if it triggers more frequently than anticipated by its minimum inter-arrival time (MIT), which could cause enormous problems in a system with hard deadlines. Two methods are possible in preventing overruns: for hardware sporadic interrupts, inhibit the interrupt with device control registers at certain times; or use a sporadic server. The sporadic server uses a timer event to turn interrupts back on after an MIT-amount of time, after turning interrupts off and releases an entry to allow a sporadic task to start execution. An exception can be raised instead of releasing the task, if software events need to be handled.

Detecting Resource Overuses

Errors in resource access are hard to handle, as they can corrupt shared state and cause liveness problems. The use of priority inheritance and ceiling protocols for control can cause problems if blocking time assumed by schedulability analysis is incorrect.
Categories of resource overuses: task overrun allocated access time for a resource; unexpected resource contention missed from schedulability analysis (common in prewritten library code).
Why timeouts don't work: with inheritance protocols, blocking is cumulative, while timeouts can be used on entry to critical sections, keeping track of total blocking time is resource intensive, and generating overheads; with immediate ceiling protocols, the blocking occurs before execution, making timeouts useless; certain elements in Ada, like protected procedures, have no timeout mechanisms due to their atomicity needs.

Group Budgets

In addition to sporadic events, aperiodic events causes further problems as they have no MIT guaranteed. If mishandled, they can cause other tasks to miss guaranteed deadlines. Aperiodic servers are needed to give priority to periodic and then sporadic tasks, while allowing aperiodic tasks to execute as much as possible when requested.
However, since Ada does not directly support servers, a better mechanism called **group budgets** can be used to allow a group tasks to share a CPU-time budget. If the group execution budget runs out, a handler can be fired to correct the conditions.
Group budget requirements: a task can only be the member of one group, a group budget can only apply to a single processor. Current budget must be non-negative. Tasks are not directly terminated when budget exhausts, but the user needs to program the appropriate action. Terminated tasks are automatically removed from any group.
A group-budgeted task first registers with a protected object that manages the budget, it then loops around waiting for the next invocation event. The group budget's handler holds all member tasks if the budget is exhausted. The handler for a budget timer replenish a fixed amount of budget at regular intervals, and release any holding tasks at replenishment.



Task Level Error Recovery

In a hard real-time task, each task can be assigned two WCET values: a higher value for conservative schedulability analysis, and a lower value for overrun detection. On detection, a simple recovery can be performed, so that the higher value is the sum of the lower value and the WCET of recovery procedure. If WCET overrun happens in a soft real-time task, it may be ignored if isolation prevents

it from affecting the schedulability of hard real-time tasks. Alternatively, its priority can be reduced, or operation re-attempted by a new release.
A task can be first allowed to run for a certain amount of time (lower value), if it executes more than this amount of time, it will have its priority temporarily lowered and run for another small amount of time. If this time limit is again exceeded, then the task must be terminated through an ATC.
For sporadic events, different violation policies such as exception, ignore and replace can be used to perform different operations on an MIT violation.
For firm real-time tasks, because it produces no value beyond its deadline, it can be simply terminated when overrunning.

System Level Error Recovery
If a task in isolation cannot deal with the problem, reconfiguration and mode changes may need to be performed. This may involve altering or terminating tasks, starting new tasks, allocating more time to critical computations. Early-termination may mean tasks must immediately return less precise results or be prepared to restart with new inputs.

Conclusions
Ada is a mature real-time technology that has obtained a niche market for high-integrity systems, but has failed capture a broader real-time and embedded systems market, which is dominated by C and C++ but has low productivity. Java seems to increase productivity but lacks in real-time support, which is being worked on by Real-time Java (RTSJ).