HRT-HOOD†

A Structured Design Method for Hard Real-time Systems‡

A. Burns and A.J. Wellings

Real-time and Distributed Systems Research Group, Department of Computer Science
University of York, Heslington, York, YO1 5DD, UK

Email: burns@minster.york.ac.uk, andy@minster.york.ac.uk

Keywords: Structured Design Methods, Schedulability Analysis, Dependability

ABSTRACT

Most structured design methods claim to address the needs of hard real-time systems. However, few contain abstractions which directly relate to common hard real-time activities, such as periodic or sporadic processes. Furthermore, the methods do not constrain the designer to produce systems which can be analysed for their timing properties.

In this paper we present a structured design method called HRT-HOOD (Hard Real-Time Hierarchical Object Oriented Design). HRT-HOOD is an extension of HOOD, and includes object types which enable common hard real-time abstractions to be represented. The method is presented in the context of a hard real-time system life cycle, which enables issues of timeliness and dependability to be addressed much earlier on in the development process. We argue that this will enable dependable real-time systems to be engineered in a more cost effective manner than the current practise, which in effect treats these topics as performance issues.

To illustrate our approach we present a simple case study of a Mine Drainage Control System, and show how it can be designed using the abstractions presented in the paper.

1. Introduction

The most important stage in the development of any real-time system is the generation of a consistent design that satisfies an authoritative specification of requirements. Where real-time systems differ from the traditional data processing systems is that they are constrained by certain non-functional requirements (e.g. dependability and timing). Typically the standard structured design methods do not cater well for expressing these types of constraints[Kopetz1991].

The objective of this paper is to present a structured design method which is tailored towards the construction of real-time systems in general, and hard real-time systems in particular. We use the term hard real-time systems to mean those systems which have components which must produce

---

† HOOD is a trademark of the HOOD User Group

‡ The work has been supported, in part, by the European Space Agency (ESTEC Contract 9198/90/NL/SF) and by the UK Defence Research Agency (Contract Number 2191/023).

This paper has been accepted for publication in the Real-time Systems Journal.

timely services; failure to produce a service within the required time interval may result in severe damage to the system or the environment, and may potentially cause loss of life (for example in avionics systems). Rather than developing a new method from scratch, the HOOD method is used as a baseline. The new method, called HRT-HOOD (Hard Real-Time HOOD), was designed as part of an European Space Agency (ESA) supported project. HOOD was chosen as the base-line because ESA currently recommend the use of HOOD for their systems development. However, we believe the ideas presented in the paper can be used to extend other common design methods such as Mascot[Simpson1986]. Although hard real-time systems can be designed using structured methods such as HOOD and Mascot, these methods lack explicit support for common hard real-time abstractions. Consequently, their use is error prone and can lead to systems whose real-time properties cannot be analysed. HRT-HOOD, in contrast, constrains the system decomposition so that the final design in amenable to timing analysis using such techniques as fixed priority or earliest deadline scheduling.

A design method cannot be presented in isolation but must be considered within the context of the overall system life cycle of which it is a part. Unfortunately the traditional system life cycle also fails to address adequately the needs of the hard real-time systems designer. Section 2 of this paper therefore gives an overview of the real-time systems design process and, in Section 3, the HRT-HOOD systems life cycle, which embraces hard real-time needs, is presented. The logical and physical architectural design activities are identified as being the key areas where hard real-time issues must be addressed. Section 4 considers the logical architecture activity and identifies the abstractions needed to represents hard real-time systems. Section 5 then addresses how the logical design can be mapped onto the physical resources of the system. Topics such as schedulability and dependability analysis are considered at this stage. Section 6, presents the HRT-HOOD method which addresses the key issues introduced in sections 4 and 5. Section 7 presents a simple case study which illustrates the approach. Section 8 briefly compares HRT-HOOD with other structured design methods. Finally in section 9 we present our conclusions.

Object Oriented Design

Throughout this paper we shall use an object abstraction for the design method. In particular we will use an extended HOOD object framework[Agency1991]. HOOD attempts to combine the advantages of object-oriented design and hierarchical decomposition. It therefore supports three basic software engineering principles[Agency1991]:

1) Abstraction, information hiding and encapsulation

   An object is defined by the service it provides to its users; the internal details are hidden. The services it provides are specified in the object's interface. The internal details are described in *OPeration Control Structure* (OPCS) procedures, and its behaviour in a synchronisation agent called an *OBject Control Structure* (OBCS).

2) Hierarchical decomposition

   Parent objects may be decomposed into child objects; HOOD uses the term "include" to represent the parent-child relationship.

3) Control structuring

   Operations on objects are activated by control flows (threads). In general there may be several threads operating simultaneously in an object. HOOD uses the term "use" to indicate that one object requires the services of another.

A HOOD object has *static* and *dynamic* properties. The static properties are defined to be the

object's interface, and the internal components of the object which implements the functionality implied by the interface. HOOD defines the dynamic properties of an object by describing the effect on the *calling* object of invoking an operation. These are:

- *Sequential* flow: where control is transferred directly to the required operation. The flow of control is described within the internals of the operation (the OPCS). After completion, control is returned in the calling object.
- *Parallel* flow: where control is *not* transferred directly to the called object but an *independent* flow is activated in the *Object Control Structure* (OBCS) of the called object. This activation takes the form of an execution request for the required operation. The reaction of the object to this request will depend on the internal state of the called object. As with sequential flow, the OPCS for each operation defines the logic of the operation.

For a more detailed discussion of HOOD see the HOOD reference manual[Agency1991] or Robinson[Robinson1992].

2. Overview of the HRT-HOOD Design Process

It is increasingly recognised that the role and importance of non-functional requirements in the development of complex critical applications has hitherto been inadequately appreciated[Burns1991c]. Specifically, it has been common practice for system developers, and the methods they use, to concentrate primarily on functionality and to consider non-functional requirements comparatively late in the development process. Experience shows that this approach fails to produce safety critical systems. For example, often timing requirements are viewed simply in terms of the performance of the completed system. Failure to meet the required performance often results in ad hoc changes to the system. This is not a cost effective process.

Non-functional requirements include dependability[Laprie1989] (e.g. reliability, availability, safety and security), timeliness (e.g. responsiveness, orderliness, freshness, temporal predictability and temporal controllability), and dynamic change management[Kramer1988] (i.e. incorporating evolutionary changes into a non-stop system). These requirements, and the constraints imposed by the execution environment, need to be taken into account throughout the system development life cycle.

We believe that if hard real-time systems are to be engineered to high levels of dependability, the real-time design method must provide:

- the explicit recognition of the types of activities/objects that are found in hard real-time systems (i.e. cyclic and sporadic activities);
- the integration of appropriate scheduling paradigms with the design process;
- the explicit definition of the application timing requirements for each object;
- the explicit definition of the application reliability requirements for each object;
- the definition of the relative importance (criticality) of each object to the successful functioning of the application;
- the support for different modes of operation — many systems have different modes of operation (e.g., take-off, cruising, and landing for an aircraft); all the timing and importance characteristics will therefore need to be specified on a per mode basis;
- the explicit definition and use of resource control objects;
- the decomposition to a software architecture that is amenable to processor allocation, schedulability and timing analysis;

- facilities and tools to allow the schedulability analysis[Kligerman1986] to influence the design as early as possible in the overall design process;
- tools to restrict the use of the implementation language so that worst case execution time analysis can be carried out;
- tools to perform the worst case execution time and schedulability analysis.

In the next section we show how these key aspects can be addressed by extending the usual software development life cycle.

3. The HRT-HOOD Software Development Life Cycle

Most traditional software development methods (including HOOD) incorporate a life cycle model in which the following activities are recognised:

- Requirements Definition — during which an authoritative specification of the system's required functional and non-functional behaviour is produced. It is beyond the scope of this paper to address issues of requirements capture and analysis. We assume that the techniques for identifying objects and their operations are those described in the HOOD User Manual[Agency1989b].
- Architectural Design — during which a top-level description of the proposed system is developed.
- Detailed Design — during which the complete system design is specified.
- Coding — during which the system is implemented.
- Testing — during which the efficacy of the system is tested.

For hard real-time systems this has the significant disadvantage that timing problems will only be recognised during testing, or worse after deployment.

A constructive way of describing the process of system design is as a progression of increasingly specific *commitments*[Dobson1990, Burns1990a]. These commitments define properties of the system design which designers operating at a more detailed level are not at liberty to change. Those aspects of a design to which no commitment is made at some particular level in the design hierarchy are effectively the subject of *obligations* that lower levels of design must address. Early in design there may already be commitments to the structure of a system, in terms of object definitions and relationships. However, the detailed behaviour of the defined objects remains the subject of obligations which must be met during further design and implementation.

The process of refining a design — transforming obligations into commitments — is often subject to *constraints* imposed primarily by the execution environment (Figure 1). The execution environment is the set of hardware and software components (e.g. processors, task dispatchers, device drivers) on top of which the system is built. It may impose both resource constraints (e.g. processor speed, communication bandwidth) and constraints of mechanism (e.g. interrupt priorities, task dispatching, data locking). To the extent that the execution environment is immutable these constraints are fixed.

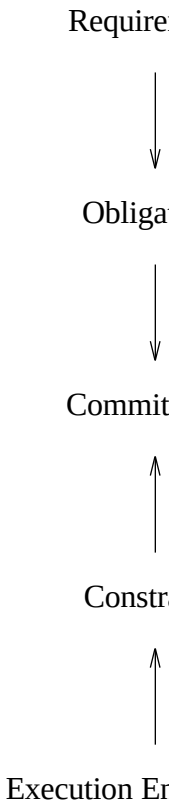


Figure 1: Obligations, Commitments and Constraints

Obligations, commitments and constraints have an important influence on the architectural design of any application. We therefore define two activities of the architectural design[Burns1991c]:

- the logical architecture design activity;
- the physical architecture design activity.

The logical architecture embodies commitments which can be made independently of the constraints imposed by the execution environment, and is primarily aimed at satisfying the functional requirements (although the existence of timing requirements, such as end-to-end deadlines, will strongly influence the decomposition of the logical architecture). The physical architecture takes these functional requirements and other constraints into account, and embraces the non-functional requirements. The physical architecture forms the basis for asserting that the application's non-functional requirements will be met once the detailed design and implementation have taken place. It addresses timing and dependability requirements, and the necessary schedulability analysis that will ensure (guarantee) that the system once built will function correctly in both the value and time domains (within some failure hypotheses). To undertake this analysis it will be necessary to make some initial estimations of the execution time of the proposed code (and other resource requirements such as LAN usage), and to have available the time dependent behaviour of the target processor and other aspects of the execution environment. Dependability analysis evaluates the design with respect to reliability, safety and security.

Although the physical architecture is a refinement of the logical architecture its development

will usually be an iterative and concurrent process in which both models are developed/modified. The analysis techniques embodied in the physical architecture can, and should, be applied as early as possible. Initial resource budgets can be defined that are then subject to modification and revision as the logical architecture is refined. In this way a 'feasible' design is tracked from requirements through to deployment.

Once the architectural design activities are complete, the detailed design can begin in earnest and the code for the application produced. When this has been achieved, the execution profile of the code must again be estimated (using a worst case execution time analyser tool) to ensure that the initial estimated worst case execution times are indeed accurate. If they are not (which will usually be the case for a new application), then either the detailed design must be revisited (if there are small deviations), or the designer must return to the architectural design activities (if serious problems exist)†. If the estimation indicates that all is well, then testing of the application proceeds. This should involve measuring the actual time of the code's execution. The modified life cycle is presented in Figure 2.

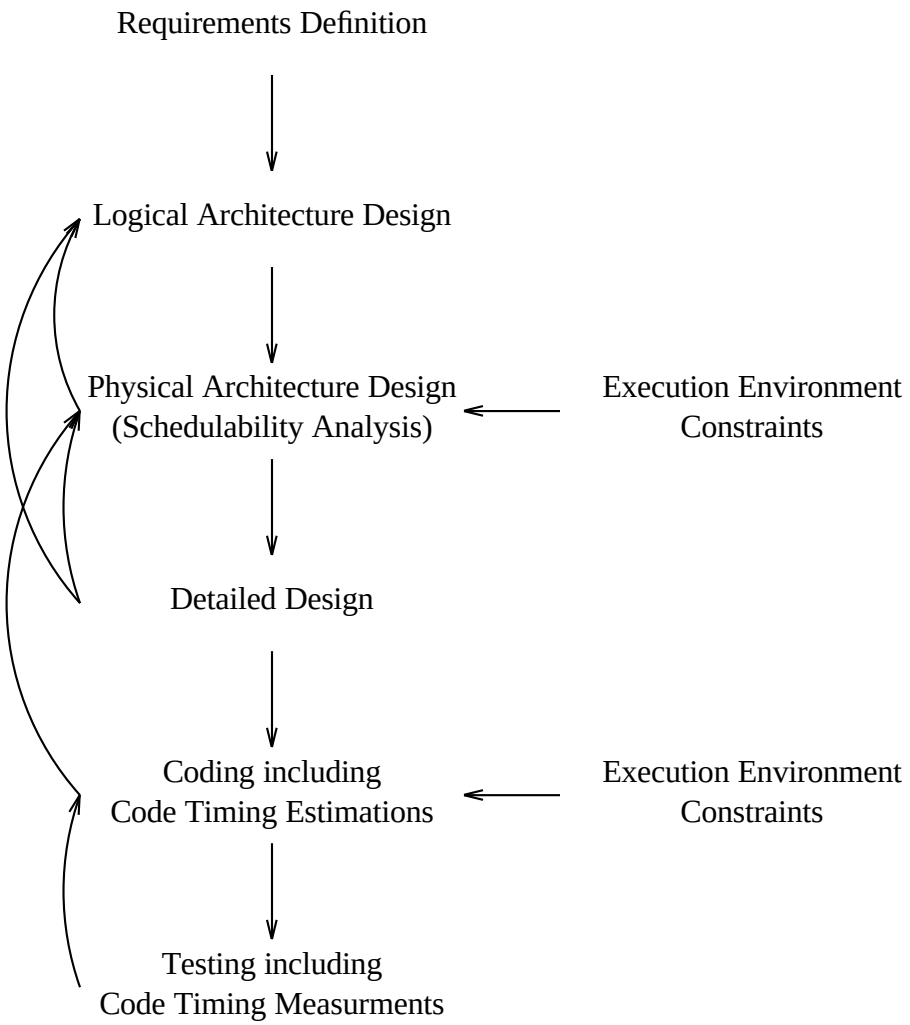


Figure 2: The Hard Real-time Life Cycle

† Of course as the detailed design progresses, it may be necessary to revisit the logical and physical architectures to provide new or modified functionality (possibly as a result of a change in requirements).

Detailed Design and Coding should follow the usual process, although code measurement for worst case timing behaviour is a complex issue. It will be necessary to constrain the way in which code is written so that analysis of execution time can be carried out in a way that is not too pessimistic. For example all loops must be bounded, and there must be limited recursion only.

4. Logical Architecture Design

There are two aspects of any design method which facilitate the logical architecture design of hard real-time systems. Firstly, explicit support must be given to the abstractions that are typically required by hard real-time system designers. *We take the view that if designs are to be well structured so that they can be analysed, then it is better to provide specific design guidelines rather then general design abstractions.* For example, supporting the abstraction of a periodic activity allows the structure of that activity to be visible to the design process which, in turn, facilitates its analysis. In contrast, allowing the designer to construct periodic activities out of some more primitive "task" activity produces designs which are more difficult to analyse. Clearly, care must be taken to ensure that the design method does not become cluttered with too many abstractions, however, it is important that the design method allows the expression of important analysable program structures.

The second aspect involves constraining the logical architecture so that it can be analysed during the physical architecture design activity. In particular designs are forced to map down to a computational model which facilitates analysis. For example, the model supports single threaded objects which interact via data-oriented communication and synchronisation mechanisms (rather than via tightly synchronous communication).

These aspects are now discussed in detail.

4.1. Supporting Common Hard Real-time Abstractions

The outcome of the logical architecture design activity is a collection of terminal objects (objects which do not require further decomposition) with all their interactions fully defined. It is assumed that some form of functional decomposition process has led to the definition of these terminal objects. Although this decomposition is essentially functional, the existence of timing requirements is a major driver in the construction of the application's architecture. Timing requirements are usually derived from other (higher-level) requirements such as controllability. They normally take the form of deadlines between input and output actions. These input/output relationships also represent functional behaviour, and hence the initial decomposition of the system can be along functional lines.

In HOOD only two basic object types are defined: PASSIVE and ACTIVE. HRT-HOOD extends the number of base types to include: PROTECTED, CYCLIC and SPORADIC. They are defined as follows.

- PASSIVE — objects which have no control over when invocations of their operations are executed, and do not spontaneously invoke operations in other objects.
- ACTIVE — objects which may control when invocations of their operations are executed, and may spontaneously invoke operations in other objects. ACTIVE objects are the most general class of objects and have no restrictions placed on them.
- PROTECTED — objects which may control when invocations of their operations are executed, and do not spontaneously invoke operations in other objects; in general PROTECTED objects may *not* have arbitrary synchronisation constraints and must be analysable for the blocking

times they impose on their callers.

- CYCLIC — objects which represent periodic activities, they may spontaneously invoke operations in other objects, but the only operations they have are requests which demand immediate attention (they represent asynchronous transfer of control, ATC, requests).
- SPORADIC — objects which represent sporadic activities; SPORADIC objects may spontaneously invoke operations in other objects; each sporadic has a *single* operation which is called to invoke the sporadic, and one or more operations which are requests which demand immediate attention (they represent asynchronous transfer of control requests).

HRT-HOOD distinguishes between the synchronisation required to execute the operations of an object and any internal independent concurrent activity within the object. The synchronisation agent of an object is called the Object Control Structure (OBCS). The concurrent activity within the object is called the object's THREAD. The thread executes independently of the operations, but when it executes operations the order of the executions is controlled by the OBCS.

A hard real-time program designed using HRT-HOOD will contain at the terminal level only CYCLIC, SPORADIC, PROTECTED and PASSIVE objects. ACTIVE objects, because they cannot be fully analysed, will only be allowed for background activity. ACTIVE object types may also be used during decomposition of the main system but must be transformed into one of the above types before reaching the terminal level.

CYCLIC and SPORADIC activities are common in real-time systems; each contains a single *thread* that is scheduled at run-time. PROTECTED objects control access to data that is shared by more than one thread (i.e. CYCLIC or SPORADIC object); in particular they provide mutual exclusion. PROTECTED objects are similar to monitors[Hoare1974] and conditional critical regions[Brinch-Hansen1973] in that they can block a caller if the conditions are not correct for it to continue. A PASSIVE object is either used by only one other object or when it can be used concurrently without error.

With these types of terminal objects other common paradigms used in hard real-time systems can be supported, in particular precedence constrained activities. These involve a series of computations through terminal objects. They are likely to occur in a design which must reflect *transaction* deadlines. For example, consider a reactive system which on receipt of a value from its input sensors must produce an output value to an actuator within a certain period. The production of the actuator setting might require the coordination of more than one objects (often in the form of precedence constraints). The term transaction is used here to represent the totality of computation and communication required to produce the output.

At a high level of design, a transaction is represented by a single object: CYCLIC or SPORADIC depending on whether the transaction is time driven or event driven. The object is then decomposed into a set of terminal precedence constrained objects. If the first activity of the transaction is cyclic then there is a *cyclic precedence constrained activity* (or *cyclic transaction*); if the first activity were sporadic (e.g., event triggered) then it would be a *sporadic precedence constrained activity* (or *sporadic transaction*). The key property of precedence constrained activities is that any activity within it can start immediately its predecessor has terminated. Hence, there is a collection of *before* and *after* relationships.

In summary, the logical architecture design process may commence with the production of ACTIVE and PASSIVE objects, and by a process of decomposition will lead to the production of terminal objects of the appropriate character. Transactions are therefore represented as CYCLIC or SPORADIC objects which decompose to terminal objects with precedence constrained and related

activities.

4.2. Constraining the Design for Analysis

In order to analyse the full design certain constraints are required. These are mainly concerned with the allowed communication/synchronisation between objects. They are:

(a) CYCLIC and SPORADIC objects may not call arbitrary blocking operations in other CYCLIC or SPORADIC objects.

(b) CYCLIC and SPORADIC objects may call operations which effect an asynchronous transfer of control operations in other CYCLIC or SPORADIC objects.

(c) PROTECTED objects may not call blocking operations in any other object.

(d) PASSIVE operations contain only sequential code which does not need to synchronise with any other object.

Where appropriate the design method itself prohibits the above behaviours.

Asynchronous transfer of control is used to get the *immediate* attention of a thread. The details of how it is implemented is, however, a concern for the implementation language and the underlying operating system (or kernel). It can be used during fast mode changes but is not a general requirement; many applications will not make use of the facility.

Other constraints may be placed on the design to aid dependability analysis. For example, to facilitate object replication it may be necessary to ensure that a particular object is deterministic.

It is important to emphasise that any hard real-time design method must constrain the design process if it is to produce analysable software. HRT-HOOD is explicitly designed to ensure that system decomposition conforms to a set of constraints that facilitates analysis of the final system.

5. Physical Architecture Design

The physical architectural design activity is concerned with mapping the logical architecture onto the required physical resources in the target system†. In order to undertake the necessary analysis, we must:

- have a design expressed in a manner that facilitates analysis
- have a means of predicting the behaviour of the design on a given platform (hardware and kernel).

The logical architecture design activity ensures that the design conforms to a computational model which facilitates timing analysis. The exact form this analysis must take is not defined by HRT-HOOD. We have, however, successfully integrated the HRT-HOOD design process with the use of static priority analysis and preemptive dispatching[Audsley1993, Burns1993a, Burns1993b]. The construction of cyclic or best-effort scheduling would also be possible with the computational model. In general physical architecture design is concerned with four activities:

1) object allocation — the allocation of objects in the logical architecture to processors within the constraints imposed by the functional and non-functional requirements (e.g, ensuring that all device controller objects are located on the sites where the controlled devices reside, or that all replicas run at separate sites)

---

† For some designs the emphasis of the physical architectural design activity may be on determining the minimal resources required to meet all the timing and dependability requirements. The ideas presented in this section equally apply to this case.

2) network scheduling — scheduling the communications network so message delays are bounded

3) processor scheduling — determining the schedule (static or dynamic) which will ensure that all tasks within all objects residing on all processors will meet their deadlines

4) dependability — for example, determining whether an object should be replicated to tolerate hardware failures, estimating the complexity of an object to see whether software fault tolerant techniques should be employed

In general, the three activities of task allocation, processor scheduling and network scheduling are all NP-hard problems[Burns1991a]. This has led to a view that they should be considered separately. Unfortunately, it is often not possible to obtain optimal solutions (or even feasible solutions) if the three activities are treated in isolation. For example, allocating a task T to a processor P will increase the computational load on P, but may reduce the load on the communications media (if T communicates with tasks on P), and hence the response time of the communications media is reduced, allowing communications deadlines elsewhere in the system to be met. The tradeoffs can become very complex as the hard real time architecture becomes more expressive. It is beyond the scope of this paper to discuss these issues. The reader is referred to the relevant literature[Dhall1978, Bannister1983, Davari1986, Chen1990, Ramamritham1990, Wilf1986, Zhao1987, Fohler1989, Audsley1991a, Tindell1992, Xu1990].

Whatever allocation and scheduling method is used, the design method must support the definition of a physical architecture by:

1) allowing timing attributes to be associated with objects,

2) providing the abstractions with which the designer can express the handling of timing (and other run-time) errors

The physical design must of course be feasible within the context of the execution environment. This is guaranteed by the allocation and schedulability analysis.

Issues of dependability must also be addressed during this activity. Both hardware and software failures must be considered. As an example of the issues addressed in the construction of the physical architecture, suppose that the reliability requirements together with knowledge of the hardware's failure characteristics imply that a particular object be replicated (so that a copy is always available, given the system failure hypotheses). The replicated object is a single entity in the logical architecture; in the physical architecture it becomes several objects, each mapped to a particular processor. Suppose further that the chosen approach to obtaining the required level of reliability is such that the replication is 'active', that is, all copies must always have the same state. The physical architecture expresses this property, and makes explicit the obligations:

a) that the object is deterministic, and

b) that an atomic broadcast protocol must be provided or implemented.

Other forms of replication such as leader/follower[Barrett1990] or passive replication can also be specified. Each will present specific obligations.

5.1. Object Attributes

The way in which the non-functional requirements are specified during the physical architecture design activity is via object attributes. All terminal objects have associated real-time attributes. Many attributes are associated with mapping the timing requirements on to the logical design (e.g., deadline, importance). These must be set before the schedulability and dependability analysis can

be performed. Other attributes (such as priority, required replication etc) can only be set during this analysis.

For each specified mode of operation, the CYCLIC and SPORADIC objects have a number of temporal attributes defined:

- The period of execution for each CYCLIC object.
- The minimum arrival interval for each SPORADIC object.
- Offset times for related objects.
- Deadlines for all sporadic and cyclic activities.

Two forms of deadline are identified. One is applied directly to a sporadic or cyclic activity. The other is applied to a precedence constrained activity (transaction); here there is a deadline on the whole activity and hence only the last activity has a true deadline. The deadlines for the other activities must be derived so that the complete transaction satisfies its timing requirements (in all cases).

To undertake the schedulability analysis, the worst case execution time for each thread and all operations (in all objects) must be known. After the logical design activity these can be estimated (taking into account the execution environment constraints) and appropriate attributes assigned. Clearly, the better the estimates the more accurate the schedulability analysis. Good estimates come from component reuse or from arguments of comparison (with existing components on other projects). During detailed design and coding, and through the direct use of measurement during testing, better estimates will become available which typically will require the schedulability analysis to be redone.

In general not all activities within the system will be at the same level of criticality. CYCLIC and SPORADIC objects will therefore be annotated with a criticality level (e.g. safety critical, mission critical, background). As well as providing valuable information for the schedulability analysis, the criticality level may also be used during the validation and verification of the total design. Critical activities may receive more rigorous analysis of both their timing characteristics and their functional characteristics, than less critical activities. Note that although PASSIVE and PROTECTED objects do not have a directly associated criticality (importance) level, for validation and verification purposes, they will be assigned a criticality level that equals the highest of their users.

During the physical architecture design activity it will be necessary to commit to a run-time scheduling approach, for example static priority scheduling with priorities assigned using rate or deadline monotonic scheduling theory[Liu1973, Audsley1993]. This requires other attributes to be supported. For example the priority of CYCLIC and SPORADIC objects. PROTECTED objects may be implemented by having ceiling priorities assigned that are at least as high as the maximum priority of the threads that use the operations defined on that object[Rajkumar1989].

In general the physical architecture design is concerned with annotating (via attributes) the objects contained in the logical design. There are however a number of cases in which extra objects may be added during this activity. In addition to replication for availability it may be desirable to reduce output jitter (a non-functional requirement); a CYCLIC object (say) with period T and deadline D, which could produce its output anywhere between its minimal execution time and D, may be replaced by two related objects and a PROTECTED object. The first object which contains all the functionality of the original will have a deadline of D1 (D1 < D) by which time it will have placed the output data in a PROTECTED object. The second CYCLIC process, which actually performs the output and which also has a cyclic time of T, will have an offset of D1 and a deadline

of D-D1. The closer D1 is made to D the smaller the jitter.

There is a final important point to emphasise about the activity of architectural design that we have called the physical design. Although there are benefits from seeing it as a distinct activity, it will typically proceed concurrently with the logical architectural design. It may be natural to add timing requirements to objects as they are defined. However, the full analysis can only be undertaken once a complete set of terminal objects is known. Furthermore, the overheads associated with the proposed execution environment must be taken into account when estimating execution times etc.

5.2. Handling Timing (and other run-time) Errors

Schedulability analysis can only be effective if the estimations/measurements of worst case execution time is accurate. Within the timing domain two strategies can be identified for limiting the effects of a fault in a software component:

- Do not allow an object to use more computation time (budget time) than it requested.
- Do not allow an object to execute beyond its deadline.

One would expect a design method to allow a designer to specify the actions to be taken if objects overrun their allocated execution time or miss their deadlines. In both of these cases the object could be informed (via an exception) that a timing fault will occur so that it can respond to the error (within the original time frame, be it budget or deadline). There is an obligation on the execution environment to undertake the necessary time measurements and to support a means of informing an object that a fault has occurred. There is also an obligation on the coding language to provide primitives that will allow recovery to be programmed.

SPORADIC objects present further difficulties if they execute more often than was envisaged when the schedulability analysis was carried out. There is an obligation on the execution environment to ensure that a SPORADIC object does not execute too early and to allow recovery action to be taken if invocations come too frequently.

A common system paradigm that makes use of SPORADIC objects is interrupt driven I/O. The logical architecture will map the interrupt to the non-blocking start operation of a SPORADIC object. The arrival of an interrupt will therefore release the SPORADIC object. For the schedulability analysis, undertaken as part of defining the physical architecture, the interrupt will be modelled as a sporadic action with a minimum arrival interval. Interrupts can also be mapped to operations on PROTECTED objects, which may further release SPORADIC objects.

If the failure hypothesis of the hardware system is that interrupts will not occur too often then the above model is adequate. However if the software system wishes to protect itself from a fault that would be caused by an over-active interrupt source then it must disable interrupts for a defined period.

5.3. Other Forms of Analysis

The emphasis so far in this paper has been on the timing requirements of hard real-time systems. There will however be other forms of analysis that should be carried out as part of the definition of a physical architecture (even for a single processor system). If there is a fixed memory constraint then this will impose obligations on the detailed design and coding. Where power consumption is an issue then the amount of RAM must be limited, and the constructive use of ROM may need to be addressed. Even weight constraints may have an influence on the physical architecture (this would be particularly true if distribution or multiprocessors were an option). We believe, however, that the

framework introduced in this section can be extended into these areas.

6. Hard Real-time HOOD (HRT-HOOD)

In this section we describe in detail the abstractions provided by HRT-HOOD, and illustrate their graphical representation. We first consider the object types and then the object attributes of HRT-HOOD.

6.1. HRT-HOOD Objects

Figure 3 illustrates the graphical representation of an HRT-HOOD object (this is almost identical to that of HOOD, the only difference being the addition of the object type indicator). There is also a textual representation of an object but this will not be discussed in this paper[Burns1991b].

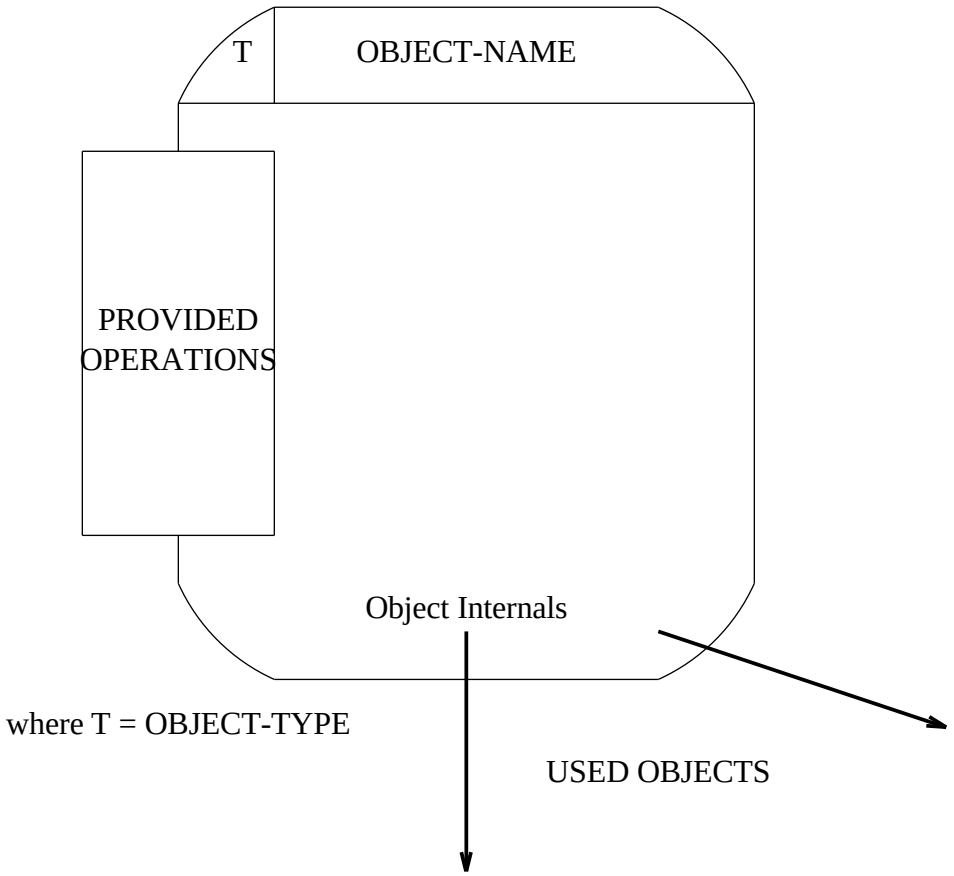


Figure 3: The Graphical Representation of an HRT-HOOD Object

6.1.1. PASSIVE

PASSIVE objects have no control over when their operations are executed. That is, whenever an operation on a passive object is invoked, control is immediately transferred to that operation. Each operation contains only sequential code which does not synchronise with any other object (i.e. it does not block). A PASSIVE object has no OBCS and no THREAD.

6.1.2. ACTIVE

The operations on an ACTIVE object may be constrained or unconstrained. Unconstrained operations are executed as soon as they are requested, similar to the operations on a PASSIVE object. They may be used to access an object's read-only data or its unprotected shared data.

Constrained operations are executed under the control of the OBCS. As with HOOD, there are two classes of constraints that can affect when requested operations are executed and their effect on the calling object.

1) *Function activation constraints* impose constraints on when a requested operation can be executed according the object's internal state. An operation is said to be "open" if the object's internal state allows the operation's execution (for example: a buffer object would allow a "put" operation if its internal storage was not full). An operation is said to be "closed" if the object's internal state does not allow the operation's execution (for example: a buffer object may not allow a "put" operation if its internal storage is full; an operation requesting an object to stop its execution can only follow a start request). An operation which has no functional activation constraints is considered to be "open".
2) *Type of request* constraints indicate the effect on the caller of requesting an operation. The following request types are supported:

   Asynchronous Execution Request — ASER

   When an ASER operation is called, the caller is not blocked by the request. The request is simply noted and the caller returns. In a process/message-based design method this would be equivalent to asynchronous message passing.

   Loosely Synchronous Execution Request — LSER

   When an LSER operation is called, the caller is blocked by the request until the called object is ready to service the request. In a process/message-based design method this would be equivalent to the occam/CSP synchronous style of message passing.

   Highly Synchronous Execution Request — HSER

   When an HSER operation is called, the caller is blocked by the request until the called object has serviced the request. In a process/message-based design method this would be equivalent to the Ada extended rendezvous style of message passing (or remote invocation[Burns1990b]).

   As with HOOD[Agency1991], both LSER and HSER may have an associated timeout in which case they are termed TOER_LSER (Timed Operation Execution Request LSER) or TOER_HSER.

In general a constrained operation may have a functional activation constraint, and will always have a type of request constraint.

ACTIVE objects in HRT-HOOD are identical to those in HOOD.

6.1.3. PROTECTED

PROTECTED objects are used to control access to resources which are used by hard real-time objects. The intention is that their use should constrain the design so that the run-time blocking for resources can be bounded (for example by using priority inheritance[Sha1990], or some other limited blocking protocol such as the immediate priority ceiling inheritance associated with the Ada9X protected records[Intermetrics1991]).

PROTECTED objects are objects which are able to control access to their operations, but

(unlike ACTIVE objects) they do not necessarily require independent threads of control. A PROTECTED object does have an OBCS but this is a monitor-like construct: operations access the internal data under mutual exclusion, and functional activation constraints may be placed on when operations can be invoked. For example, a bounded buffer might be implemented as a PROTECTED object.

Two types of constrained operations are available on PROTECTED objects:

- *Protected synchronous execution request* (PSER)
- *Protected asynchronous execution request* (PAER).

A constrained operation can only execute if no other constrained operation has access to the protected data.

Only a PSER type of request can have a functional activation constraint. A timeout PSER request (TOER_PSER) must have a functional activation constraint.

PROTECTED objects may also have non-constrained operations, which are executed in the same manner as PASSIVE operations.

6.1.4. CYCLIC

CYCLIC objects are used to represent periodic activities. They are active objects in the sense that they have their own independent threads of control. However, these threads (once started) execute irrespective of whether there are any outstanding request for their objects' operations. Furthermore, they do not wait for any of their objects' operations at any time during their execution. Indeed, in many cases CYCLIC objects will not have any operations.

In general CYCLIC objects will communication and synchronise with other hard real-time threads by calling operations in PROTECTED objects. However, it is recognised that some constrained operation may be defined by a CYCLIC object because:

- other objects may need to signal a mode change to the cyclic object — this could be achieved by having CYCLIC objects polling a "mode change notifier" PROTECTED object but this is inefficient if the response time required from the CYCLIC object is short (if mode changes can occur only at well defined instances then "mode change notifier" objects would be appropriate)
- other objects may need to signal error conditions to the cyclic object — this could again be achieved by having CYCLIC objects polling an "error notifier" PROTECTED object but this is again inefficient when the response time required from the CYCLIC object is short

Several types of constrained operations are therefore available on CYCLIC objects (each may have functional activation constraints). All of these, when open, require an immediate response from the CYCLIC's thread. The OBCS of a CYCLIC object interacts with the thread to force an asynchronous transfer of control. Available operations include an:

- *Asynchronous, asynchronous transfer of control request* (ASATC). This is similar to the ASER for active objects except that it demands that the CYCLIC object reacts "immediately". The request will result in an asynchronous transfer of control in the CYCLIC object's thread. Note that the call is also asynchronous in that the caller is not blocked waiting for the transfer of control to take place. Hence the call is an *asynchronous, asynchronous transfer of control request*. Other forms of ATC could be provided, for example a loosely synchronous ATC (LSATC) or a highly synchronous ATC (HSATC).

  A CYCLIC object may start its execution immediately it is created, it may have an offset (that

is a time before which the THREAD should be delayed before starting its cyclic execution), or it may synchronise its start via a PROTECTED object.

All CYCLIC objects have a thread, whereas only those with operations have an OBCS.

6.1.5. SPORADIC

SPORADIC objects are active objects in the sense that they have their own independent threads of control. Each SPORADIC object has a single constrained operation (usually named START) which is called to invoke the execution of the thread. The operation is of the type which does not block the caller (ASER), it may be called by an interrupt, in which case the label becomes ASER_BY_IT. The operation which invokes the sporadic has a defined minimum arrival interval, and/or a maximum arrival rate.

A SPORADIC object may have other constrained operations but these are requests which wish to affect immediately the SPORADIC to indicate a result of a mode change or an error condition. As with CYCLIC objects ASATC operations are possible. A SPORADIC object which receives a asynchronous transfer of control request will immediately abandon it current computation.

Each terminal SPORADIC object has a single thread, and an OBCS.

6.2. Real-time Object Attributes

HOOD does not explicitly support the expression of many of the constraints necessary to engineer real-time systems. In the object description skeleton language (ODS) there is a field in which the designer can express "implementation and synchronisation" constraints. Rather than use this to express an object's real-time attributes, a separate REAL-TIME ATTRIBUTES field has been added. These attributes are normally filled in at the *terminal* object level. It is anticipated that many of the values of the attributes will be computed by support tools.

The following attributes represent the type of information a designer might wish to express about an application. Some of them may depend on the scheduling theory being used or the approach to fault tolerance.

- DEADLINE

  Each CYCLIC and SPORADIC object may have a defined deadline for the execution of its thread.

- OPERATION_WCET

  Each externally visibly operation of an object must have a worst case execution time defined.

- OPERATION_BUDGET

  Each externally visibly operation of an object may have a budget execution time defined.

  If an operation has an allocated budget there must be an internal operation declared within the object which can be called by the run-time support system if the budget is overrun. The worst case execution time for the external operation is the operation's budget time plus the budget time of the internal error handling operation.

- THREAD_WCET

  Each CYCLIC and SPORADIC object must have a worst case execution time defined for its thread of execution.

- THREAD_BUDGET

  Each CYCLIC and SPORADIC object may have a budget execution time defined for each

activation of its thread of execution. An overrun of the budgeted time may result in the activity being undertaken by the thread being terminated. Each CYCLIC and SPORADIC object must therefore have an internal operation which is to be called if its thread's budget time is overrun. The worst case execution time for the thread is the thread's budget time plus the budget time of the internal error handling operation.

- PERIOD

  Each CYCLIC object must have a defined period of execution.

- OFFSET

  Each CYCLIC object may have a defined offset which indicate the time that the THREAD should delay before starting its cyclic operations.

- MINIMUM_ARRIVAL_TIME or MAXIMUM_ARRIVAL_FREQUENCY

  Each SPORADIC object must have either a defined minimum arrival time for requests for its execution, or a maximum arrival frequency of request.

- PRECEDENCE CONSTRAINTS

  A THREAD may have precedence constraints associated with its execution.

- PRIORITY

  Each CYCLIC and SPORADIC object may have a defined priority for its thread. This priority is defined according to the scheduling theory being used.

- EXECUTION_TRANSFORMATION

  A CYCLIC or a SPORADIC object may need to be transformed at run-time to incorporate extra delays. This may be required, for example, as a result of period transformation during the schedulability analysis phase of the method.

- IMPORTANCE

  Each CYCLIC and SPORADIC object must have a defined importance for its thread. This importance represents whether the thread is a hard real-time thread or a soft real-time thread.

This list may be extended — for example some hard real time scheduling theories require minimum/average execution times, utility/benefit functions etc; fault tolerant systems my require objects to be identified for replication; safety critical systems may also require integrity levels to be assigned.

6.3. The HRT-HOOD Use Relationship

As with HOOD, PASSIVE objects must not use constrained operations of other objects, and ACTIVE objects may use operations of any other object freely. Furthermore, HRT-HOOD forbids PASSIVE objects to use each other in a cyclic manner.

The use relationships for the other object types are as follows.

- CYCLIC and SPORADIC objects must not use constrained operations of terminal ACTIVE objects unless they are asynchronous; they can, however, use constrained operations of non-terminal ACTIVE objects if these operations are implemented by child PROTECTED objects. CYCLIC and SPORADIC objects may use constrained operations of PROTECTED objects

  In any hard real-time system the blocking time of a thread must be bounded. Consequently operation calls which may result in arbitrary blocking must be prohibited. In the method, hard real-time threads synchronise with each other via the constrained operations on PROTECTED objects. Terminal ACTIVE objects are not considered by the method to be hard real-time, and

consequently CYCLIC and SPORADIC objects must not be dependent upon their execution.

- CYCLIC and SPORADIC objects may call the constrained operations of other CYCLIC or SPORADIC objects.

  These operations may result in some blocking. The worst case response time of an asynchronous transfer of control request to each CYCLIC object should be definable. Timeouts may be used to bound this time if the worst case response is unacceptable.

- PROTECTED objects may only use ASER constrained operation or constrained operations on other PROTECTED objects as long as they have no functional activation constraints.

  In general PROTECTED objects should not block once they are executing (although some requeuing of requests may be allowed).

The following table summarises the HRT-HOOD use relationship. Note that any object may use an unconstrained operation of any other object.

Caller	CYCLIC	SPORADIC	PROTECTED	ACTIVE	PASSIVE
CYCLIC	✓	✓	✓	ASER only	✓
SPORADIC	✓	✓	✓	ASER only	✓
PROTECTED	ASATC only	ASATC and START	✓	ASER only	✓
ACTIVE	✓	✓	✓	✓	✓
PASSIVE	✗	✗	✗	✗	✓

6.4. The HRT-HOOD Include Relationship (decomposition)

The "include relationship" for HRT-HOOD objects are as follows:

- An ACTIVE object may include any other object.
- A PASSIVE object may only include other PASSIVE objects.

  Note that in HOOD a PASSIVE object may include an ACTIVE object as long as the passive nature of the parent is not violated. HRT-HOOD removes this feature as it is no longer necessary, given HRT-HOOD's increase in expressive power.

- A PROTECTED object may include PASSIVE objects, and one PROTECTED object.

  The intention is that PROTECTED objects should not have separate threads of control. Consequently they can not decompose into objects which have their own independent threads. Furthermore, a parent PROTECTED object guarantees that its data is accessed under mutual exclusion; any decomposition must not violate the parent's guarantee.

- A SPORADIC object may include at least one SPORADIC object along with one or more PASSIVE, and PROTECTED objects.

  At one level of the design an object may be considered SPORADIC. However, it may be decomposed into a group of precedenced constrained SPORADIC objects (communicating via PROTECTED objects) as long as the SPORADIC nature of the parent is not violated.

- A CYCLIC object may include at least one CYCLIC object along with one or more PASSIVE, PROTECTED and SPORADIC objects.

  At one level of the design an object may be considered CYCLIC. However, it may be decomposed into a group of precedenced constrained CYCLIC, PASSIVE, PROTECTED and SPORADIC objects as long as the CYCLIC nature of the parent is not violated.

The HRT-HOOD "include" rules are summarised in the table below.

Parent	CYCLIC	SPORADIC	PROTECTED	ACTIVE	PASSIVE
CYCLIC	✓	✓	✓	✗	✓
SPORADIC	✓	✓	✓	✗	✓
PROTECTED	✗	✗	✓	✗	✓
ACTIVE	✓	✓	✓	✓	✓
PASSIVE	✗	✗	✗	✗	✓

6.5. Operation Decomposition

One parent constrained operation may be *implemented by* one child constrained operation. The graphical representation is a dashed (or shaded) arrow from the parent operation to the child operation.

A constrained operation shall be implemented by a constrained child operation. The following decompositions are valid (an * indicates an operation with a functional activation constraint):

Decomposition	Comments
ASER → ASER ASER → ASATC	a parent asynchronous operation can be implemented by a child asynchronous operation, or an asynchronous asynchronous transfer of control operation
ASER → PSER ASER → PAER	a parent asynchronous operation can be implemented by a child synchronous† or asynchronous protected operation
*ASER → *ASER *ASER → PSER *ASER → PAER *ASER → *ASATC	an asynchronous request must never block
LSER → LSER LSER → PSER LSER → *PSER LSER → LSATC	a parent loosely synchronous operation can be implemented by a child loosely synchronous operation, a protected synchronous request, a protected synchronous with a functional activation constraint, or a loosely synchronous asynchronous transfer of control operation
*LSER → *LSER *LSER → *PSER *LSER → *LSATC	similar to the above
HSER → HSER HSER → PSER HSER → *PSER HSER → HSATC	a parent highly synchronous operation can be implemented by a child highly synchronous operation, a protected synchronous request, a protected synchronous with a functional activation constraint, or a highly synchronous asynchronous transfer of control operation
*HSER → *HSER *HSER → *PSER *HSER → *HSATC	similar to the above
PSER → PSER *PSER → *PSER PAER → PAER	constrained operations on a parent PROTECTED object must be implemented by identical operations in the child.

Note, that all the constrained operations of a parent PROTECTED object must be implemented by the same child PROTECTED object, in order to guarantee the mutual exclusion of the parent.

---

† Note that this mapping is allowed because on a single/multi processor the time taken to execute the operation on a PROTECTED object is assumed to be small and therefore in effect asynchronous. For a distributed system this decomposition would force the two objects onto the same processing node.

6.6. Translation of the Design into the Implementation Language

It is beyond the scope of this paper to consider in detail the translation of HRT-HOOD designs into an implementation language. However, the following gives an approximation of the translation in terms of general programming language constructs.

passive object	module/package with each operation as a procedure/function
active object	module/package with each operation as a procedure/function and a set of internal tasks and a synchronisation agent
protected object	module/package with each operation as a procedure/function and a synchronisation agent
cyclic object	module/package with each ATC operation as a procedure/function, a synchronisation agent and a periodic process/task; the synchronisation agent can asynchronously affect the flow of control in the periodic process/task
sporadic object	module/package with each ATC and the START operation as a procedure/function, a synchronisation agent and a process/task; the synchronisation agent can asynchronously affect the flow of control in the periodic process/task; for the start operation, the sporadic process/task executes one iteration; alternative mappings may be provided for sporadics which are invoked by an interrupt

For a detailed discussion of mapping HRT-HOOD to Ada 9X see Burns and Wellings[Burns1992]. Appendix A illustrates this mapping for the "environment monitor" object given in the case study below.

6.7. Summary

In this section we have described the abstractions provided by HRT-HOOD. These abstraction have been chosen because they are precisely those abstraction which system designers need when dealing with hard real-time systems. The decomposition rules, as noted earlier, enforce a design which is amenable to timing analysis.

In the following section we present a case study which illustrates the HRT-HOOD design method.

7. An Illustrative Case Study

The example that has been chosen is based on one which commonly appears in the literature[Kramer1983, Sloman1987, Shrivastava1987, Burns1990b, Burns1990a] (and one that the authors have already investigated using PAMELA[Burns1990b]); it possesses many of the characteristics which typify embedded real-time systems. For simplicity, it is assumed that the system will be implemented on a single processor. We shall also focus of the timing analysis only, and assume that static priorities and Deadline Monotonic Scheduling Theory[Audsley1993, Leung1982] is to be used for analysing the timing properties of the system.

7.1. Mine Drainage

The study concerns the design of software necessary to manage a simplified pump control system for a mining environment. The system is used to pump mine-water, which collects in a sump at the bottom of the shaft, to the surface. A simple schematic diagram illustrating the system is given in Figure 4.

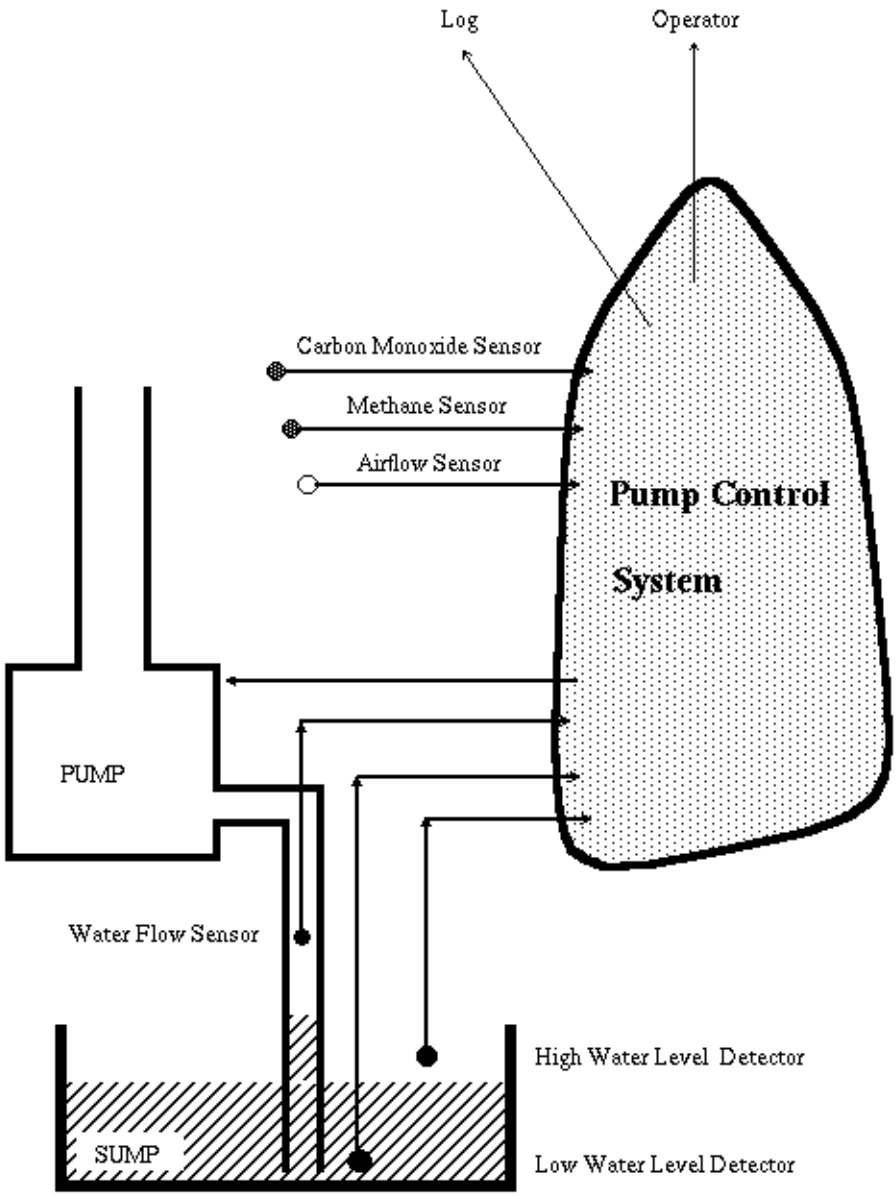


Figure 4: A Mine Drainage Control System

The relationship between the control system and the external devices is shown in Figure 5. Note that only the high and low water sensors communicate via interrupts (indicated by shaded arrows); all the other devices are either polled or directly controlled.

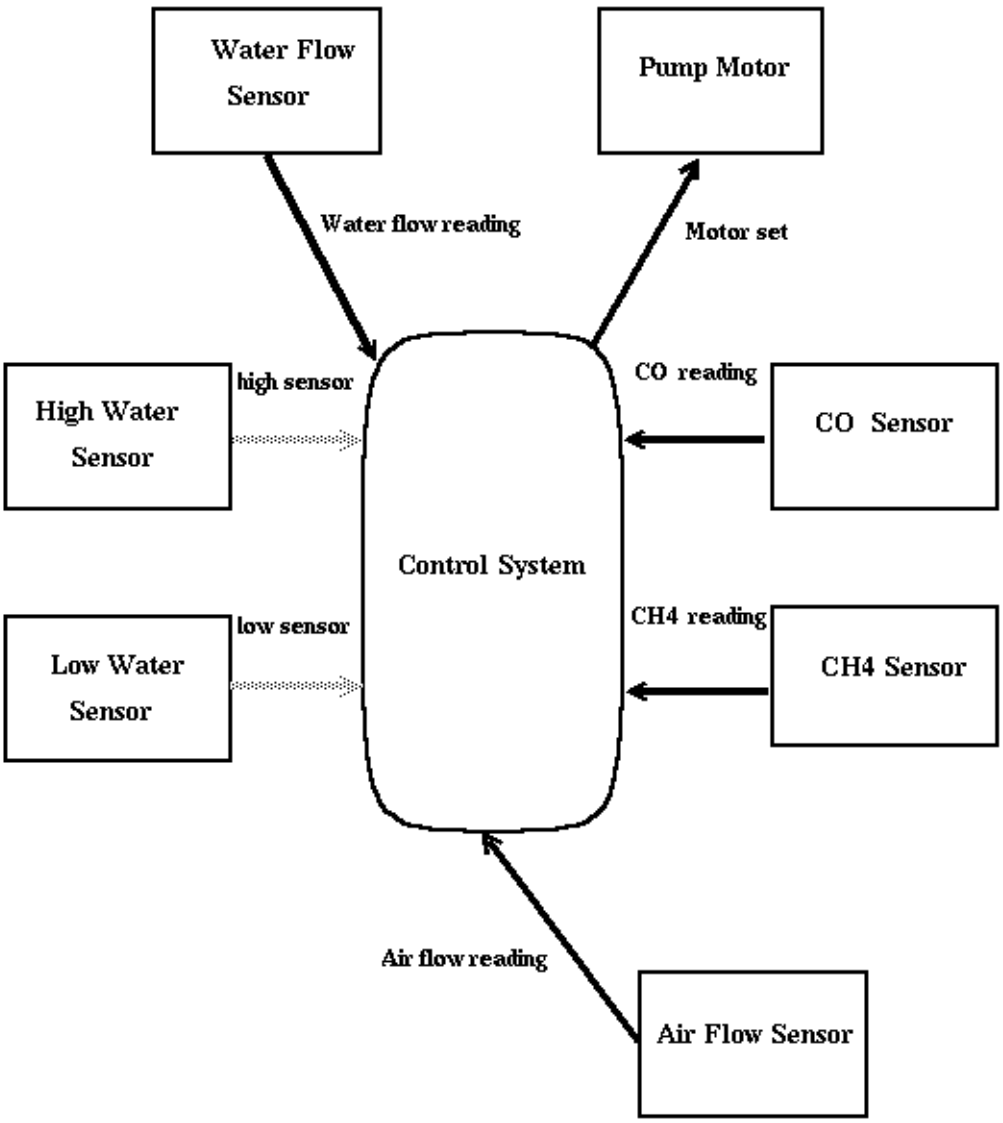


Figure 5: Graph Showing External Devices

7.2. Functional Requirements

The functional specification of the system may be divided into four components: the pump operation, the environment monitoring, the operator interaction, and system monitoring.

Pump operation

The required behaviour of the pump is that it monitors the water levels in the sump. When the water reaches a high level (or when requested by the operator) the pump is turned on and the sump is drained until the water reaches the low level. At this point (or when requested by the operator) the pump is turned off. A flow of water in the pipes can be detected if required.

The pump should only be allowed to operate if the methane level in the mine is below a critical level.

Environment monitoring

The environment must be monitored to detect the level of methane in the air; there is a level beyond which it is not safe to operate the pump. The monitoring also measures the level of carbon monoxide in the mine and detects whether there is an adequate flow of air. Alarms must be signalled if gas levels become critical.

Operator interaction

The system is controlled from the surface via an operator's console. The operator is informed of all critical events.

System monitoring

All the system events are to be stored in an archival database, and may be retrieved and displayed upon request.

7.3. Non-functional Requirements

The non-functional requirements can be divided into three components: timing, dependability, and security. This case study is mainly concerned with the timing requirements and consequently we shall not address dependability and security (see Burns and Lister[Burns1991c] for a full consideration of dependability and security aspects).

There are several requirements which relate to the timeliness of system actions. The following list is adapted from Burns and Lister[Burns1991c]:

(i) Monitoring periods

The maximum periods for reading the environment sensors (see above) may be dictated by legislation. For the purpose of this example we assume these periods are the same for all sensors, namely 60 seconds. In the case of methane there may be a more stringent requirement based on the proximity of the pump and the need to ensure that it never operates when the methane level is critically high. This is discussed in (ii) below.

We assume that the water level detectors are event driven, and that the system should respond within 20 seconds.

(ii) Shut-down deadline

To avoid explosions there is a deadline within which the pump must be switched off once the methane level exceeds a critical threshold. This deadline is related to the methane sampling period, to the rate at which methane can accumulate, and to the margin of safety between the level of methane regarded as critical and the level at which it explodes. The relationship can be expressed by the inequality

$$R(P + D) < M$$

where

R is the rate at which methane can accumulate
P is the sampling period
D is the shut-down deadline
M is the safety margin.

Note that the period P and the deadline D can be traded off against each other.

In this example we shall assume that the presence of methane pockets may cause levels to rise rapidly, and therefore we require a sampling period of 5 seconds and a shut-down deadline of 1 second.

(iii) Operator information deadline

The operator must be informed: within 1 second of detection of critically high methane or carbon monoxide readings, within 2 seconds of a critically low air-flow reading, and within 3 seconds of a failure in the operation of the pump.

In summary the sensors have the following defined periods or minimum inter-arrival rates (in seconds) and deadlines.

	periodic/sporadic	arrival times	deadline
CH4 sensor	P	5.0	1
CO sensor	P	60.0	1
water flow sensor	P	60.0	3
airflow sensor	P	60.0	2
high/low water interrupt handler (HLW handler)	S	100.0	20

Figure 6: Attributes of Periodic and Sporadic Processes

7.4. The HRT-HOOD Logical Architecture Design

We now develop a logical architecture for the pump control system. In the logical architecture we address those requirements which are independent of the physical constraints (eg. processor speeds) imposed by the execution environment. The functional requirements identified in the previous section fall into this category. Consideration of the other system requirements is deferred until design of the physical architecture, described later.

First Level Decomposition

The first step in developing the logical architecture is the identification of appropriate classes of object from which the system can be built. The functional requirements of the system suggest four distinct subsystems (ACTIVE objects):

(i) *pump controller subsystem*, responsible for operating the pump;

(ii) *environment monitor subsystem*, responsible for monitoring the environment;

(iii) *operator console subsystem*, the interface to human operators;

(iv) *data logger subsystem*, responsible for logging operational and environmental data.

Figure 7 illustrates this decomposition.

The pump controller has four operations: The operations "not safe" and "is safe" are called by the environment monitor to indicate to the pump controller whether it is safe to operate the pump (due to the level of methane in the environment). The "request status" and "set pump" operations are called by the operator console. As an additional reliability feature, the pump controller will always check that the methane level is low before starting the pump (by calling "check safe" in the environment monitor). If pump controller finds that the pump cannot be started (or that the water does not appear to be flowing when the pump is notionally on) then it raises an operator alarm.

The environment monitor has the single operation "check safe" which is called by the pump controller.

The operator console has the alarm operation, which as well as being called by the pump controller, is also called by the environmental monitor if any of its readings are too high. As well as receiving the alarm calls, the operator console can request the status of the pump and attempt to override the high and low water sensors by directly operating the pump. However in the latter case the methane check is still made, with an exception being used to inform the operator that the pump cannot be turned on.

The data logger has six operations which are merely data logging actions which are called by the pump controller and the environment monitor.

It should be noted that the symbol **O—>** is used to illustrate data flow, and **—|—** represents an exception.

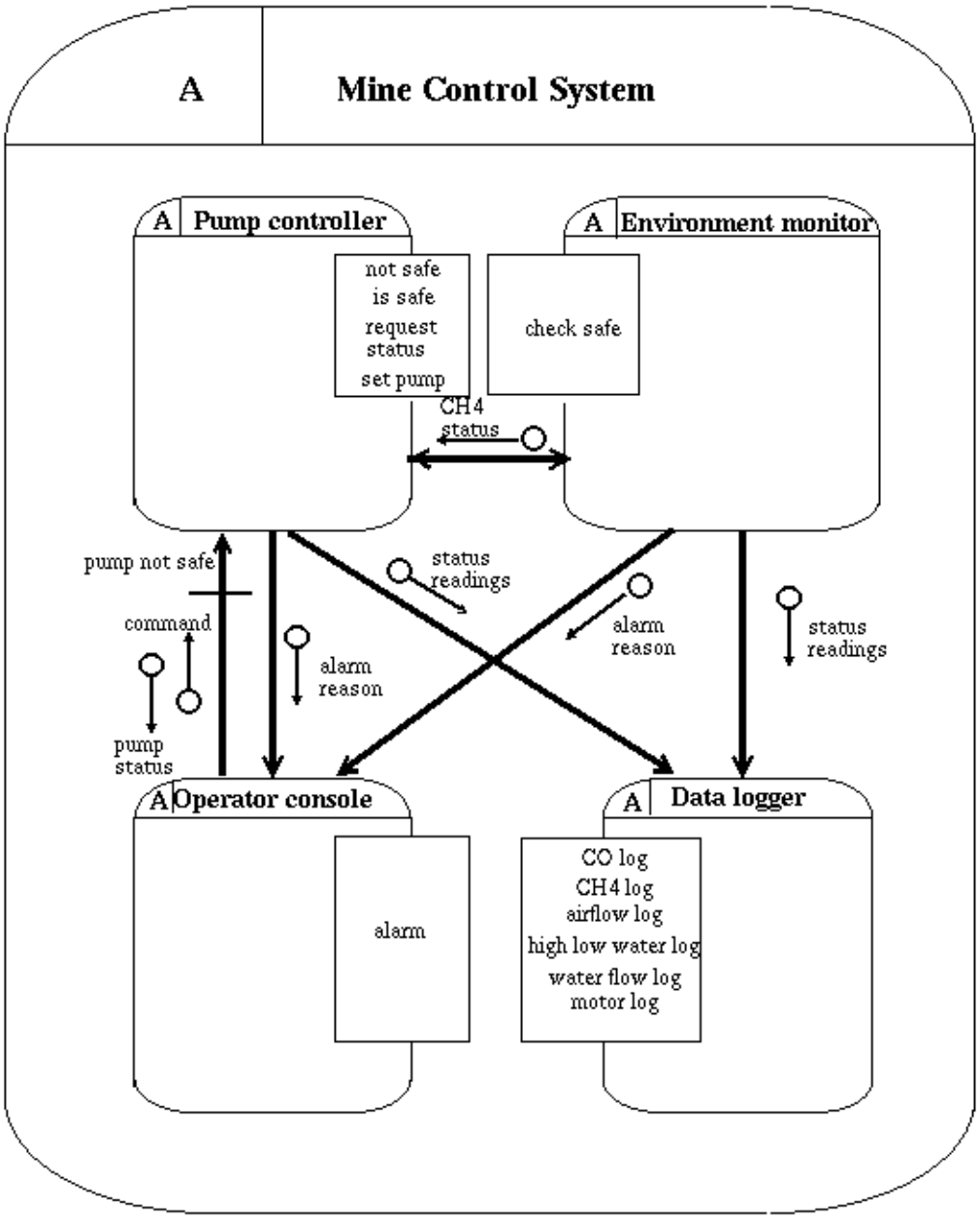


Figure 7: First Level Hierarchical Decomposition of Control System

Pump Controller

The decomposition appropriate to the pump controller is shown in Figure 8. The following symbol represents a constrained operation. The letters indicate the type of constraint.

HSER

The pump controller is decomposed into three objects. One object controls the pump motor. This is a PROTECTED object because: it simply responds to commands, requires mutual exclusion for its operations, and does not spontaneous call other objects. All of the pump controller's operations are

implemented by the motor object. As the system is real-time, we have attempted to minimise the blocking time of all operations. In this case none of the operations can be blocked (other than for mutual exclusion). The motor object will make calls to all its uncle objects.

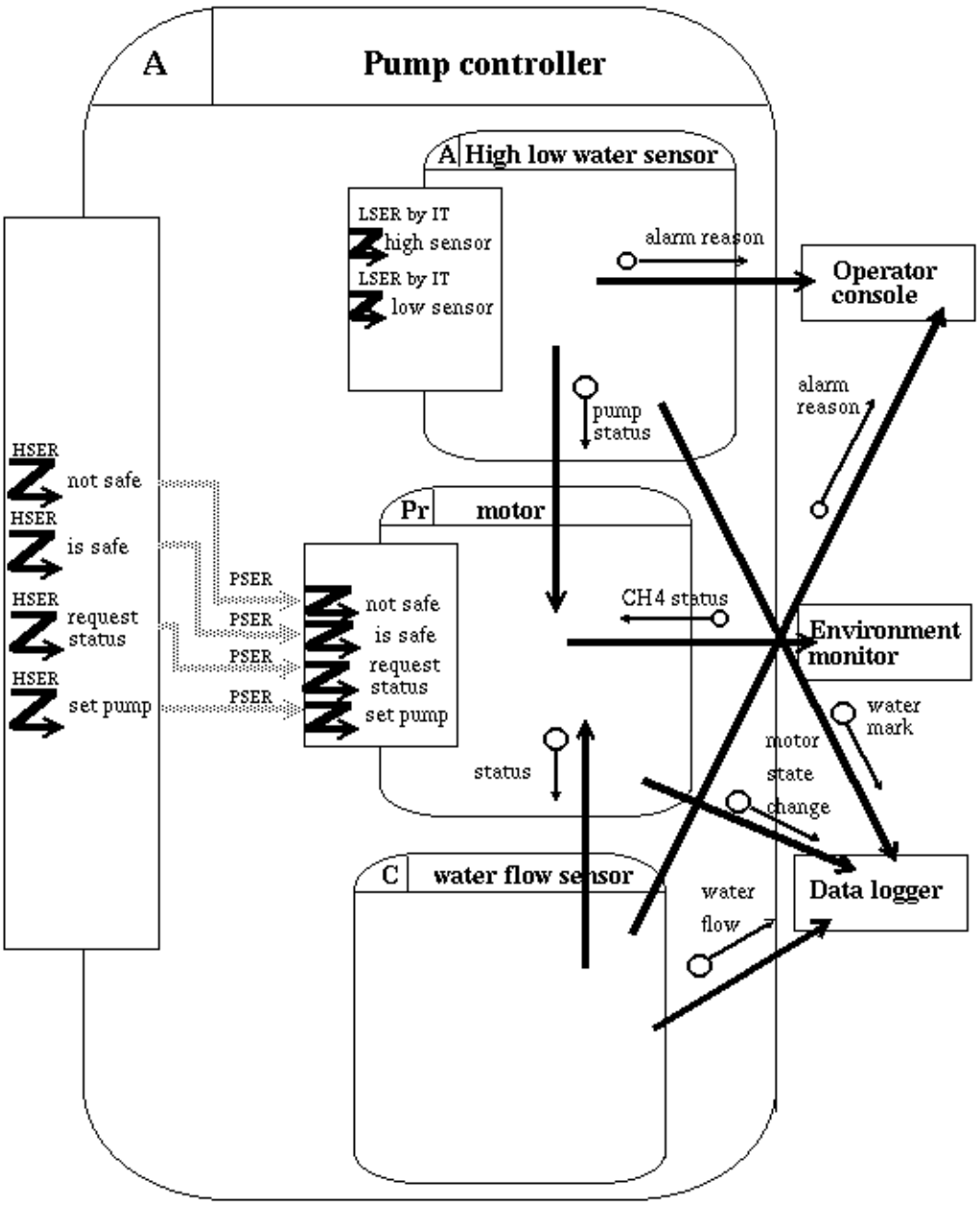


Figure 8: Hierarchical Decomposition of the Pump Object

The other two objects control the water sensors. The water flow sensor object is a CYCLIC object which continually monitors the flow of water from the mine. The high-low-water sensor is an ACTIVE object which handles the interrupts from the associated sensors. It decomposes into a PROTECTED and a SPORADIC object, as shown in Figure 9.

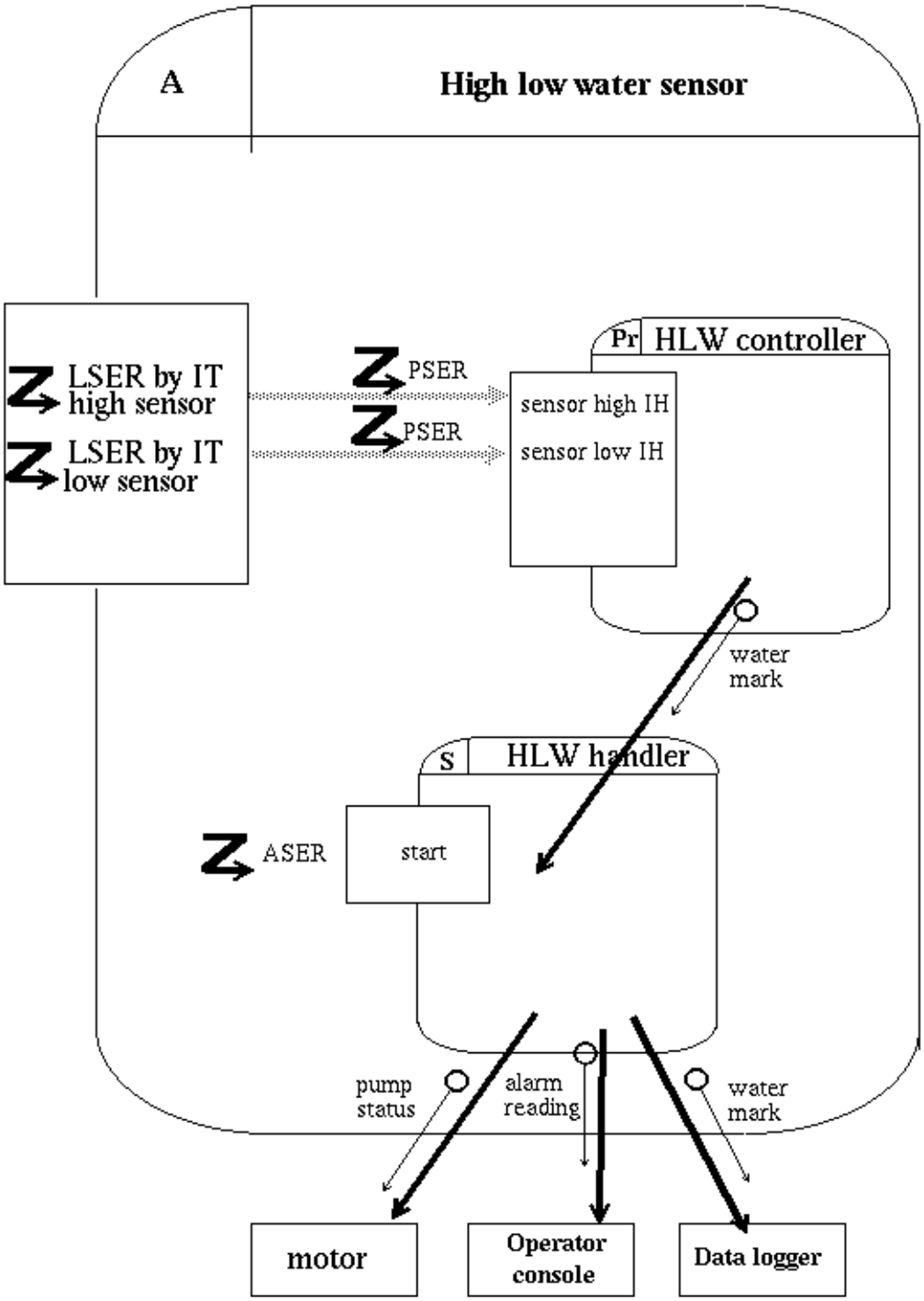


Figure 9: Decomposition of the High Low Water Sensor

The Environment Monitor

The Environment Monitor decomposes into four terminal objects, as shown in Figure 10. Three of the objects are CYCLIC objects which monitor the: CH4 level, CO level and the airflow in the mine environment. Only the CH4 level is requested by other objects in the system, consequently a PROTECTED object is used to control access to the current value.

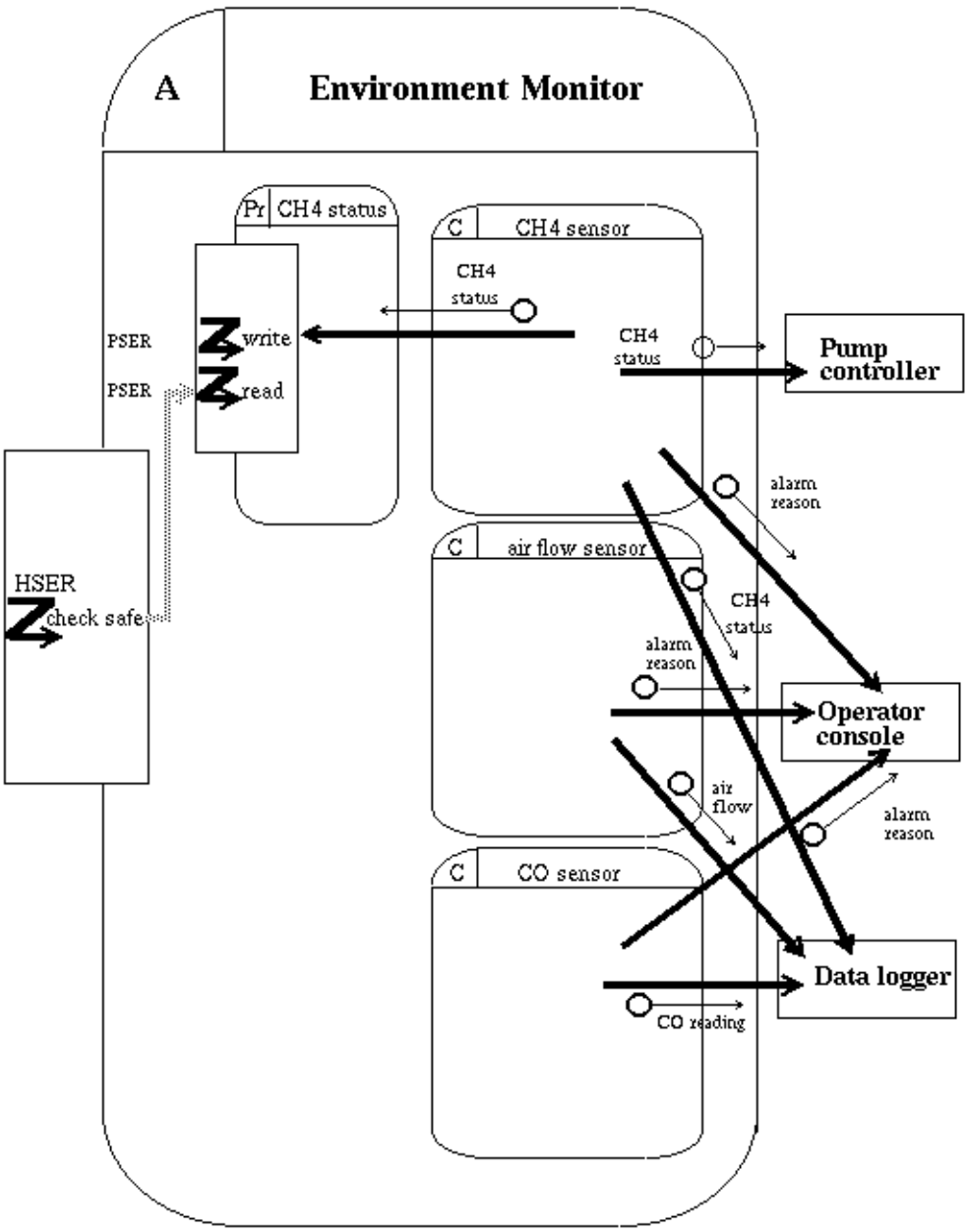


Figure 10: Hierarchical Decomposition of the Environment Monitor

Note that the HSER request of the environment monitor has been implemented by a PSER request in the CH4 status object, and hence the calling objects suffers bounded blocking.

The Data Logger and the Operator Console

We are not concerned with the details of the data logger in this case study. However, we do require that they each present an asynchronous interface.

7.5. The Physical Architecture

HRT-HOOD supports the design of a physical architecture by:

1) allowing timing attributes to be associated with objects,
2) providing a framework from within which a schedulability analysis of the terminal objects can be undertaken, and
3) providing the abstractions with which the designer can express the handling of timing errors.

The non-functional timing requirements identified in section 7.3 are transformed into annotations on methods and threads as follows.

(i) Periodicity

The threads in the environment monitor which read the carbon monoxide, water flow and air-flow sensors have periods of 60 seconds. The methane monitor thread has a period of 5 seconds in order that it can turn off a running pump within the required deadline.

(ii) Deadlines

The threads which read the environment sensors have a range of deadline (from 1 to 3 seconds) for reporting critical readings to the operator. There is also a deadline of 1 second for executing the pump controller method which switches the pump off when a critical methane level is detected. Figure 11 summarises the timing requirements.

	periodic/sporadic	arrival times	deadline
CH4 sensor	P	5.0	1
CO sensor	P	60.0	1
water flow sensor	P	60.0	3
airflow sensor	P	60.0	2
HLW handler	S	100.0	20

Figure 11: Summary of the Timing Characteristics of Threads

Execution Time Analysis

For the purpose of the case study we will assume the following execution times (in practice these would be first estimated and then derived from considering the pseudo code which had been developed during the detailed design for each object). For simplicity we shall ignore the handling of timing errors.

operation	budget time (seconds)	error handling time	wcet (seconds)
motor			
not safe	0.05	0	0.05
is safe	0.05	0	0.05
request status	0.05	0	0.05
set pump	0.10	0	0.10
water flow sensor			
thread	0.15	0	0.15
controller			
sensor high IH	0.01	0	0.01
sensor low IH	0.01	0	0.01
HLW handler			
start	0.01	0	0.01
thread	0.10	0	0.10
CH4 status			
read	0.01	0	0.01
write	0.01	0	0.01
CH4 sensor			
thread	0.25	0	0.25
airflow sensor			
thread	0.15	0	0.15
CO sensor			
thread	0.15	0	0.15

Figure 12: Summary of the Timing Characteristics of Operations

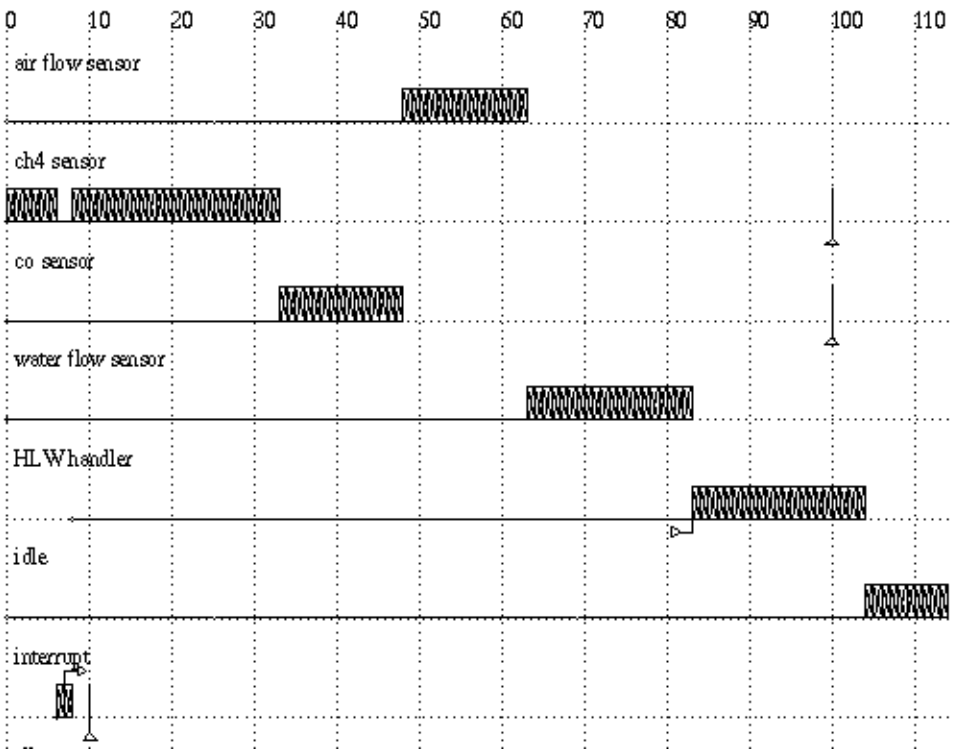
We assume that the high and low water sensor interrupts occur at a priority higher than any software task, and that the low level interrupt handling time (in HLW controller) is 0.01 seconds. All other system overheads, such as context switch times, have been incorporated into the thread and protected objects worst case execution time (see Burns and Wellings[Burns1993c] for a discussion on how the schedulability analysis can be extended to include clock overheads).

As a result of running a deadline monotonic analyser tool we determine that the system is schedulable on a single processor with the following priority settings (10 is highest priority) assuming that priority level 8 represents the hardware priorities of the high low water mark interrupts

HLW handler obcs	10
HLW controller	9
hardware interrupts	8
CH4 status	7
motor	6
CH4 sensor	5
CO sensor	4
airflow sensor	3
water flow sensor	2
HLW handler	1

Figure 13: Summary of the Priorities

Assuming all threads are released at time 0, and we have preemptive priority-based run-time scheduling, the following diagram (produced by a scheduling simulator called STRESS[Audsley1991b]) illustrates the execution of the threads. The first 100 ticks are where there is most contention, and it can be seen that all threads comfortably meet their deadlines.



This diagram shows the following aspects of a thread set simulation:

- Thread Release - circle on the time line of a thread (see airflow sensor at time 0).
- Releasing a thread blocked on a PROTECTED object - left corner of a square with a triangle (see interrupt at t = 7).
- Start of execution of a thread previously blocked on a PROTECTED object - right corner of a square with a triangle (see handler at t = 82).
- Thread Execute - a hatched box on the time line of a thread (see CH4 sensor at time 0 to 6).
- Preempted Thread - when a higher priority thread runs and a lower priority thread has remaining computation time, a dotted line extend along the time line of the thread (see CH4 sensor at time 6 to 8).
- Thread deadline - a vertical line with a triangle at its base (see CH4 sensor at t = 100).
- Missed Deadline - raised circle above the time line of a thread (not shown in the diagram)

Figure 14: Simulated Execution of the Mine Control System

Once the physical architecture design has been completed, the detailed design can begin in earnest. A full description of the detailed design and along with a systematic translation into Ada 9X is given by Burns and Wellings[Burns1991b]. An example of the translation is given in Appendix A.

8. Comparison with other Structured Design Methods

There are many structured design methods which are targeted toward real-time systems: MASCOT, JSD, Yourdon, MOON, HOOD, DARTS, MCSE etc. None of these, however, support directly the common abstractions of period and sporadic activities which are found in most hard real-time

systems. Neither do they impose a computational model that will ensure that effective timing analysis of the final system can be undertaken. For a comparison of these see Hull et al[Hull1991], Cooling[Cooling1991], and Calvez[Calvez1993]. PAMELA (Process Abstraction Method for Embedded Large Application)[Cherry1986] is perhaps the exception. It allows for the diagrammatic representation of cyclic activities, state machines and interrupt handlers. However, the notation is not supported by abstractions for resources and therefore designs are not necessarily amenable to schedulability analysis. See Burns and Wellings[Burns1990b] for a description of the Mine Control case study (presented in Section 7) in PAMELA.

There are a few CASE environments available to support the real-time systems design process; Teamwork[Technologies1990], for example, provides an environment in which designs can be expressed using Buhr's graphical notation[Buhr1984] or HOOD. Tools which analyse the timing properties of the resulting designs, however, are not included. EPOS[Lauber(Ed)1990] is another environment which supports the whole real-time system life cycle. It provides three specification languages: one for describing the customer requirements, one for describing the system specification, and one for describing project management, configuration management and quality assurance. Like HRT-HOOD, the system specification has a graphical and a textual representation. Both periodic and sporadic activities can be represented (but they are not directly visible when manipulating the diagrams) and timing requirements can be specified. Furthermore, early recognition of the real-time behaviour of applications has been addressed within the context of EPOS[Lauber1989]; however this work relies on animating the system specification rather than schedulability analysis.

One project which has addressed the issues raised by this paper is MARS[Damm1989]. In a recent paper Kopetz et al[Kopetz1991] present a design method that is influenced by a real-time transaction model. A transaction is triggered by an event (usually occurring in the environment of the system) and results in a corresponding response. During design a transaction is refined into a sequence of subtransactions, finally becoming a set of tasks which may be allocated to processing units (called "clusters"). Attributes such as deadline, criticality etc can be associated with a transaction. A system life cycle is proposed which carries out both dependability and schedulability analysis.

9. Conclusions

In this paper we have presented an extended version of HOOD which more directly addresses the concerns of the hard real-time system designer. Direct support is given to periodic activities, sporadic activities, precedence constrained activities, and time-bounded resource controllers. Moreover, the rules governing allowable decomposition of the design ensure a final system which is analysable for its timing properties.

It should be emphasised that HRT-HOOD is a new design method and not a new version of HOOD (in the sense that it is not supported by the HOOD User Group). HRT-HOOD differs from HOOD in three main areas:

1) The number of object types has been increased to include: CYCLIC, SPORADIC and PROTECTED objects. The "include" and "use" relationships have been updated to reflect these changes.
2) Object attributes have been added to terminal objects so that these objects can be annotated with their real-time characteristics (e.g. deadline, period, worst case execution time).
3) A development process is provided which facilitates the analysis of non functional

requirements.

HRT-HOOD is a structured design notation which has both a diagrammatic and a textual notation. Examples of the diagrammatic notation have been presented in this paper, the reader is referred to Burns and Wellings[Burns1991b] for a full definition of the textual representation of HRT-HOOD objects.

The technology for implementing hard real-time systems on single processors is now relatively mature. However, supporting methods are few and far between. HRT-HOOD is one of the first structured design method which seriously attempts to address the issue of designing hard real-time systems. It is therefore inevitable that this first attempt will need refinement as experience with its use is accrued. We are currently developing tool support for the method, and undertaking industrial case studies to evaluate it effectiveness[Burns1993a].

Acknowledgement

The authors would like to thank Eric Fyfe and Chris Bailey of British Aerospace Space Systems, Paco Gomez Molinero, Tullio Vardanega and Fernando Gonzalez-Barcia of the European Space Agency (ESTEC), and John McDermid of the University of York for their comments on the material presented in this paper. We would also like to acknowledge Andrew Lister of the University of Queensland who helped lay the foundations for this work.

APPENDIX: An Example Mapping of HRT-HOOD to Ada 9X

A.1 Introduction

In this appendix we present an example of the systematic translation of HRT-HOOD designs to Ada 9X. We show a simplified version of the mapping for the "Environment Monitor" object given in section 7.4. The structure of the mappings given is based on the structure given for HOOD[Agency1989a]. Other mappings are possible.

It is inevitable that a restricted subset of Ada (be it Ada 83 or Ada 9X) will be required if a tool is to be designed that analyses Ada code for its worst case execution times. This subset *excludes the following features:*

- recursive or mutually recursive subprogram calls
- unbounded loop constructs
- dynamic storage allocation
- unconstrained arrays or types containing unconstrained arrays

Although periodic threads and offsets are implemented using an Ada delay statement, the schedulability analysis cannot cope with arbitrary delays in thread execution. Consequently we *do not allow the application programmer to use*

- the delay statement.

A.2 The Environment Monitor Object

The "Environment Monitor" object is a non-terminal object. It is translated into two Ada packages. The first defines the types and constants it introduces. The second defines the operations.

```
package environment_monitor_types is
  type ch4_reading is new integer range 0 .. 1023;
  type co_reading is new integer range 0 .. 1023;
  type methane_status is (motor_safe, motor_unsafe);
  type air_flow_status is (air_flow, no_air_flow);
  co_high : constant co_reading := 600;
  ch4_high : constant co_reading := 400;
end environment_monitor_types;

with environment_monitor_types; use environment_monitor_types;
with ch4_status; use ch4_status;
package environment_monitor is -- ACTIVE

  function check_safe return methane_status renames
      ch4_status.read; -- HSER
  -- environment_monitor:
  -- calls procedures is_safe and not_safe in pump_controller
  -- calls procedure alarm in operator_console_interface
  -- calls entries air_flow_log, CO_log, CH4_log  in data_logger

end environment_monitor;
```

Note, that as this is a non-terminal object there is no package body.

The decomposition of the environment monitor subsystem was shown in Figure 10. Note that the "check_safe" subprogram allows the pump controller to observe the current state of the methane level without blocking, via the "CH4_status" protected object. All other components are periodic activities. Here we show the mapping for the "CH4_status" and "CH4_sensor" objects.

A.3 CH4_status Object

The CH4_status object is mapped to a package with an enclosed protected type.

```
with system;  use system;
with environment_monitor_types; use environment_monitor_types;
with ch4_status_rtatt; use ch4_status_rtatt;
-- ch4_status_rtatt defines the real-time attributes of the object

package ch4_status is -- PROTECTED
  protected obcs is
    pragma priority(ch4_status_rtatt.ceiling);
    procedure write (current_status : methane_status);
    function read return methane_status;
  private
    environment_status : methane_status := motor_unsafe;
  end obcs;

  function read return methane_status renames obcs.read; -- PSER
  procedure write (current_status : methane_status)
              renames obcs.write; -- PSER
end ch4_status;

package body ch4_status is

  protected body obcs is

    procedure write (current_status : methane_status) is
    begin
      environment_status := current_status;
    end write;

    function read return methane_status is
    begin
      return environment_status;
    end read;
  end obcs;

end ch4_status;
```

A.4 CH4 Sensor Handling Object

The CH4_sensor object is mapped to two packages: the first contains the object's types and internal state variables, the second contains the thread.

```
with device_register_types; use device_register_types;
with environment_monitor_types; use environment_monitor_types;
with system; use system;
package ch4_sensor_intypes is
  ch4_present : ch4_reading;
  -- define control and status register
  -- for the CH4 ADC
  ch4csr : device_register_types.csr;
  for ch4csr use at 16#aa18#;
  -- define the data register
  ch4dbr : ch4_reading;
  for ch4dbr use at 16#aa1a#;
  jitter_range : constant ch4_reading := 40;
  now : duration;
end ch4_sensor_intypes;

package ch4_sensor is -- CYCLIC
    -- no external interface
    -- calls motor.is_safe
    -- calls motor.not_safe
    -- calls operator_console.alarm
    -- calls data_logger.ch4_log
end ch4_sensor;

with ch4_sensor_intypes; use ch4_sensor_intypes;
with environment_monitor_types; use environment_monitor_types;
with monotonic; use monotonic;
with system; use system;
with ch4_sensor_rtatt; use ch4_sensor_rtatt;
package body ch4_sensor is

  procedure opcs_periodic_code is separate;
  procedure opcs_initialise is separate;

  task thread is
    pragma priority(ch4_sensor_rtatt.thread.pri);
  end thread;
```

```
  task body thread is
    t: time;
    period : duration := ch4_sensor_rtatt.period(current_mode);
  begin
    t:= clock;
    opcs_initialise;
    loop
      opcs_periodic_code;
      t := t + period;
      delay until t;
    end loop;
  end;
end ch4_sensor;

with device_register_types; use device_register_types;
separate(ch4_sensor)
procedure opcs_initialise is
begin
  -- enable device
  ch4csr.device := d_enabled;
end opcs_initialise;
```

```
with operator_console; use operator_console;
with operator_console_types; use operator_console_types;
with pump_controller; use pump_controller;
with ch4_status; use ch4_status;
with device_register_types; use device_register_types;
with data_logger; use data_logger;
separate(ch4_sensor)
procedure opcs_periodic_code is
begin
  ch4csr.operation := set; -- start conversion
  if not ch4csr.done then
    operator_console.alarm(ch4_device_error);
  else
    -- read device register for sensor value
    ch4_present := ch4dbr;
    if ch4_present > ch4_high then
      if ch4_status.read = motor_safe then
         pump_controller.not_safe;
         ch4_status.write(motor_unsafe);
         operator_console.alarm(high_methane);
      end if;
    elsif (ch4_present < (ch4_high - jitter_range)) and
        (ch4_status.read = motor_unsafe) then
      pump_controller.is_safe;
      ch4_status.write(motor_safe);
    end if;
    data_logger.ch4_log(ch4_present);
  end if;
exception
  when others =>
    operator_console.alarm(unknown_error);
    pump_controller.not_safe;
    ch4_status.write(motor_unsafe); -- try and turn motor off
                -- before terminating
end opcs_periodic_code;
```

References

Agency1989a. Agency, European Space, ‘‘HOOD Reference Manual Issue 3.0’’, WME/89-173/JB (September 1989).

Agency1989b. Agency, European Space, ‘‘HOOD User Manual Issue 3.0’’, WME/89-353/JB (December 1989).

Agency1991. Agency, European Space, ‘‘HOOD Reference Manual Issue 3.1’’, HRM/91-07/V3.1 (July 1991).

Audsley1991a. Audsley, N. C., Tindell, K., Burns, A., Richardson, M. F. and Wellings, A. J., ‘‘The DrTee Architecture for Distributed Hard Real-Time Systems’’, *Proceedings 10th IFAC Workshop on Distributed Control Systems*, Semmering, Austria (9-11 September 1991).

Audsley1991b. Audsley, N.C., Burns, A., Richardson, M.F. and Wellings, A.J., ‘‘STRESS: A Simulator for Hard Real-Time Systems’’, RTRG 106, Department of Computer Science, University of York (1991).

Audsley1993. Audsley, N.C., Burns, A. and Wellings, A.J., ‘‘Deadline Monotonic Scheduling Theory and Application’’, *Control Engineering Practice* **1**(1), pp. 71-78 (1993).

Bannister1983. Bannister, J A. and Trivedi, K.S., ‘‘Task Allocation in Fault-Tolerant Distributed Systems’’, *Acta Informatica* **20**, pp. 261-281 (1983).

Barrett1990. Barrett, P.A., Hilborne, A.M., Verissimo, P., Rodrigues, L., Bond, P.G., Seaton, D.T. and Speirs, N.A., ‘‘The Delta-4 Extra Performance Architecture(XPA)’’, *Digest of Papers, The Twentieth Annual International Symposium on Fault-Tolerant Computing*, Newcastle, pp. 481-488 (June 1990).

Brinch-Hansen1973. Brinch-Hansen, P., *Operating System Principles*, Prentice-Hall, New Jersey (July 1973).

Buhr1984. Buhr, R.J.A., *System Design with Ada*, Prentice-Hall International (1984).

Burns1990a. Burns, A. and Lister, A. M., ‘‘An Architectural Framework for Timely and Reliable Distributed Information Systems(TARDIS): Description and Case Study’’, YCS.140, Department of Computer Science, University of York (1990).

Burns1990b. Burns, A. and Wellings, A.J., *Real-time Systems and their Programming Languages*, Addison Wesley (1990).

Burns1991a. Burns, A., ‘‘Scheduling Hard Real-Time Systems: A Review’’, *Software Engineering Journal* **6**(3), pp. 116-128 (1991).

Burns1991b. Burns, A. and Wellings, A.J., ‘‘Development of a Design Methodology’’, Task 3 Deliverable on ESTEC Contract 9198/90/NL/SF, Department of Computer Science, University of York (September 1991).

Burns1991c. Burns, A. and Lister, A. M., ‘‘A Framework for Building Dependable Systems’’, *Computer Journal* **34**(2), pp. 173-181 (1991).

Burns1992. Burns, A. and Wellings, A.J., *Hard Real-time HOOD: A Design Method for Hard Real-time Ada 9X Systems*, Towards Ada 9X, Proceedings of 1991 Ada UK International Conference, IOS Press (1992).

Burns1993a. Burns, A., Wellings, A.J., Bailey, C.M. and Fyfe, E., ‘‘The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-time System Design and

Implementation'', in *Ada sans frontieres Proceedings of the 12th Ada-Europe Conference, Lecture Notes in Computer Science*, Springer-Verlag (1993).

Burns1993b. Burns, A., Wellings, A.J., Bailey, C.M. and Fyfe, E., ''The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-time System Design and Implementation'', YCS 190, Department of Computer Science, University of York (1993).

Burns1993c. Burns, A., Wellings, A.J. and Hutcheon, A.D., ''The Impact of an Ada Run-time System's Performance Characteristics on Scheduling Models'', in *Ada sans frontieres Proceedings of the 12th Ada-Europe Conference, Lecture Notes in Computer Science*, Springer-Verlag (1993).

Calvez1993. Calvez, J.P., *Embedded Real-time Systems: A Specification and Design Methodology*, Wiley (1993).

Chen1990. Chen, G. and Yur, J., ''A Branch-and-Bound-with-Underestimates Algorithm for the Task Assignment Problem with Precedence Constraint'', *10th International Conference on Distributed Computing Systems*, pp. 494-501 (1990).

Cherry1986. Cherry, G.W., *Pamela Designers Handbook*, Thought Tools Incorporated (1986).

Cooling1991. Cooling, J.E., *Software Design for Real-time Systems*, Chapman and Hall (1991).

Damm1989. Damm, A., Reisinger, J., Schwabl, W. and Kopetz, H., ''The Real-Time Operating System of MARS'', *ACM Operating Systems Review* **23**(3 (Special Issue)), pp. 141-157 (1989).

Davari1986. Davari, S. and Dhall, S.K., ''An On-line Algorithm for Real-Time Task Allocation'', pp. 194-200 in *Proceedings 7th IEEE Real-Time Systems Symposium* (December 1986).

Dhall1978. Dhall, S.K. and Liu, C.L., ''On a Real-Time Scheduling Problem'', *Operations Research* **26**(1), pp. 127-140 (1978).

Dobson1990. Dobson, J. E. and McDermid, J. A., ''An Investigation into Modelling and Categorisation of Non-Functional Requirements'', YCS.141, Department of Computer Science, University of York (1990).

Fohler1989. Fohler, G. and Koza, C., ''Heuristic Scheduling for Distributed Real-time Systems'', Nr 6/89, Institut für Technische Informatik, Technische Universität Wien, Austria (April 1989).

Hoare1974. Hoare, C.A.R., ''Monitors - An Operating System Structuring Concept'', *CACM* **17**(10), pp. 549-557 (October 1974).

Hull1991. Hull M.E.C., O'Donoghue P.G. and Hagan B.J., ''Development methods for real-time systems'', *Computer Journal* **34**(2), pp. 164-72, Comput. J. (UK) (April 1991).

Intermetrics1991. Intermetrics, ''Draft Ada 9X Mapping Document, Volume II, Mapping Specification'', Ada 9X Project Report (August 1991).

Kligerman1986. Kligerman, E. and Stoyenko, A.D., ''Real-time Euclid: A Language for Reliable Real-time Systems'', *IEEE Transactions on Software Engineering* **SE-12**(9), pp. 941-949 (September 1986).

Kopetz1991. Kopetz H., Zainlinger R., Fohler G., Kantz H., Puschner P. and Schutz W., ''The

design of real-time systems: from specification to implementation and verification'', *Software Engineering Journal* **6**(3), pp. 72-82, Softw. Eng. J. (UK) (May 1991).

Kramer1983. Kramer, J., Magee, J., Sloman, M.S. and Lister, A.M., ''CONIC: an Integrated Approach to Distributed Computer Control Systems'', *IEE Proceedings (Part E)* **180**(1), pp. 1-10 (Jan 1983).

Kramer1988. Kramer, J. and Magee, J., ''A Model for Change Management'', *Proceedings of the IEEE Distributed Computing Systems in the '90s Conference* (September 1988).

Laprie1989. Laprie, J.C., ''Dependability: A Unified Concept for Reliable Computing and Fault Tolerance'', pp. 1-28 in *Resilient Computer Systems*, ed. Anderson, T., Collins and Wiley (1989).

Lauber(Ed)1990. Lauber(Ed), R.J., ''EPOS Overview: Short Account of the Main Featurs'', GPP Gesellschaft für ProzeBrechnerprogrammierung mbH, Kolpingring 18a, D-8024 Oberhaching bei München (January 1990).

Lauber1989. Lauber, R J, ''Forecasting Real-time Behaviour During Software Design Using a CASE Environment'', *The Journal of Real-Time Systems* **1**, pp. 61-76 (1989).

Leung1982. Leung, J.Y.T. and Whitehead, J., ''On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks'', *Performance Evaluation (Netherlands)* **2**(4), pp. 237-250 (December 1982).

Liu1973. Liu, C.L. and Layland, J.W., ''Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment'', *JACM* **20**(1), pp. 46-61 (1973).

Rajkumar1989. Rajkumar, R., Sha, L. and Lehoczky, J. P., ''An Experimental Investigation of Synchronisation Protocols'', *Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 11-17 (May 1989).

Ramamritham1990. Ramamritham, K., ''Allocation and Scheduling of Complex Periodic Tasks'', *10th International Conference on Distributed Computing Systems*, pp. 108-115 (1990).

Robinson1992. Robinson, P., *Hierarchical Object-Oriented Design*, Prentice Hall (1992).

Sha1990. Sha, L., Rajkumar, R. and Lehoczky, J. P., ''Priority Inheritance Protocols: An Approach to Real-Time Synchronisation'', *IEEE Transactions on Computers* **39**(9), pp. 1175-1185 (September 1990).

Shrivastava1987. Shrivastava, S.K., Mancini, L. and Randell, B., ''On The Duality of Fault Tolerant Structures'', pp. 19 - 37 in *Lecture Notes in Computer Science* , Springer-Verlag (1987).

Simpson1986. Simpson, H.R., ''The Mascot Method'', *Software Engineering Journal* **1**(3), pp. 103-120 (May 1986).

Sloman1987. Sloman, M. and Kramer, J., *Distributed Systems and Computer Networks*, Prentice-Hall (1987).

Technologies1990. Technologies, Cadre, *Teamwork*, 222 Richmond Street, Providence, RI 02903, USA (1990).

Tindell1992. Tindell, K., Burns, A. and Wellings, A.J., ''Allocating Real-Time Tasks (An NP-Hard Problem made Easy)'', *Real-Time Systems* **4**(2), pp. 145-165 (June

1992).

Wilf1986. Wilf, H. S., *Algorithms and Complexity*, Prentice-Hall International (1986).

Xu1990. Xu, J. and Parnas, D.L., ‘‘Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations’’, *IEEE Transactions on Software Engineering* **SE-16**(3), pp. 360-369 (March 1990).

Zhao1987. Zhao, W., Ramamritham, K. and Stankovic, J.A., ‘‘Preemptive Scheduling under Time and Resource Constraints’’, *IEEE Transactions on Computers* **36**(8), pp. 949-960 (August 1987).