VSMT-OS

Product Specification

The Designer's Handbook

A complete and integrated methodology for
real-time embedded systems software development…
the smart path to real-time, embedded software solutions.

An introduction to Virtual State Machine Technology and its approach to embedded software development. It provides an overview of the architecture and features offered by VSMT and the basic steps required when developing application software to function in a VSMT environment.

Virtual State Machine Technology

The Legal Bit

CONTENTS

CONTENTS 4

INTRODUCTION 6

TERMINOLOGY AND CONCEPTS 7

ASYNCHRONOUS EVENT PROCESSING 7
CRITERIA'S 7
INCOMING MESSAGE QUEUE 7
INTERRUPT HANDLING 7
INTERRUPT HANDLING - ISR 7
INTERRUPT HANDLING -BLIH 7
LOGICAL COMPONENT 8
MESSAGE 8
OPERATING SYSTEM KERNEL 8
PROCEDURE 8
PROCESS 8
RE-ENTRANCY 8
REAL-TIME SYSTEM 8
SCHEDULER 8
STATE TRANSITION 9
SYNCHRONOUS EVENT PROCESSING 9
SYSTEM 9
TASKS 9
UTILITY 9
VIRTUAL STATE MACHINE 9
VOLUNTARY SUSPENSION 9

WHY VSMT-OS? 10

THE 'REALITY' 10
THE 'DREAM' 10
VSMT-OS TASKS 11
VSMT TASK MODEL 11
State 0 - 'cold start' 12
State 1 - 'warm start' 13
State 2 - 'idle' 14
State 'n' - all others 14

OVERVIEW 15

MECHANISMS 16

THE KERNEL 16
INTER-PROCESS COMMUNICATION 16
TASK LIFE-CYCLE 17
PROCESS STATE HANDLING 17
UNIQUE TASK IDENTIFICATION 18
SCHEDULING CRITERIA 18
TASK RE-ENTRANCY FOR MULTIPLE COPIES 18
EXTERNAL WORLD PROCESSING 18
ASYNCHRONOUS EVENT HANDLING AND COMMUNICATION 18
TIMED EVENT MONITORING 19
INTERRUPT HANDLING 19
CODE RE-ENTRANCY 19
DATA RE-ENTRANCY 19
BUFFER HANDLING 20
ENVIRONMENT INITIALISATION 20
OS CONFIGURATION 20
'WATCHDOG' CONTROL 20

DEPENDENCIES 21

PROCESSOR OCCUPANCY 21
PRIORITISATION 21
TASK DESIGN ARCHITECTURE 21
ZERO-WAIT CONFORMANCE 21

VSM PRIMITIVE UTILISATION 21
SELF-INVOCATION 21
MINIMAL 3-STATE AWARENESS 21
SPONTANEOUS TASK INVOCATION 22

DESIGN CONSIDERATION 22

TASK DESIGN CONSIDERATIONS 22
DATA HANDLING 23
BUILD CONSIDERATIONS 23
FOREGROUND ACTIVITIES 23
BACKGROUND ACTIVITIES 23
REAL-TIME CONTROL 23

APPLICATION INDEPENDENT RUN-TIME OBJECTS (AIRO) 24

KERNEL PRIMITIVES AIRO-KE 24
REAL-TIME PRIMITIVES AIRO-RT 25
MEMORY MANAGEMENT PRIMITIVES AIRO-MM 25
QUEUE MANAGEMENT AIRO-QM 26
Queue Management Primitives (Linked Lists) 26
Queue Management Primitives (Cyclic Buffers) 26
SCANNER PRIMITIVES AIRO-SC 26
GENERAL UTILITY PRIMITIVES AIRO-GU 27
SYSTEM SECURITY PRIMITIVES AIRO-SS 27

ON-LINE TEST AND COMPLIANCE TOOLS (OTCT) 28

EVENT AND SEQUENCE TRACKING PRIMITIVES OTCT-ET 28
SYSTEM MONITORING AND PERFORMANCE PRIMITIVES OTCT-MP 28
TRACE AND FAULT CREATION PRIMITIVES OTCT-TF 28

PERFORMANCE 29

STATE EVENT HANDLING IN A MEASURE OF TIME 29
STATE EVENT HANDLING AS A MEASURE OF SOURCE CODE STATEMENTS 29
STATE EVENT HANDLING AS A MEASURE OF SCHEDULER SLOT DURATION 29

SYSTEM REQUIREMENTS 30

DOCUMENT HIERARCHY AND PRODUCT COMPONENTS 31

VSMT-OS DESIGNERS HANDBOOK 31
USER INTERFACES - AIROS 31
USER INTERFACES - OTCTS 31
LOGICAL COMPONENT SPECIFICATIONS 31
VSMT-OS SOFTWARE SOURCE 32

DOCUMENT HISTORY 34

INTRODUCTION

This section describes what VSM Technology can offer the software development environment and the type of applications which may benefit from it.

Project Managers, Product Marketing and Designers of real-time embedded systems have, no doubt, experienced the rising costs and delays that are a seemingly unavoidable characteristic of software development. As the number and complexity of the logical operations performed by the software increases, so does the opportunity for architectural and performance problems increase, whether they be speed or memory related. In addition, the cost of maintaining the software as changes are made also increases as the initial structure of the software becomes lost in the changes. If reasonably priced systems of increasing complexity are to be produced of a high standard with corresponding levels of reliability, attention must be continually focused on methods for producing software more effectively.

Virtual State Machine Technology has been developed to provide a methodology covering all stages of real-time embedded systems software development. It bases its philosophy on the concept of virtual state machines - that is the support of unlimited numbers of virtual microprocessors in a single processor environment. In accordance with the original CCITTs recommendations for the definition and implementation of software systems, VSMT has taken this a stage further to provide machine and application independent modelling and implementation techniques with complete run-time components. No matter what hardware technologies are employed or microprocessor architectures preferred, VSMT offers shortest route solutions on any development host platform with its associated tools chain.

In order to support the VSMT methodology for design and implementation, an operating environment for the target system, along with on-line testing and compliance tools, is provided. The operating environment, in its entirety is referred to as the operating system, and is made up of application independent run-time objects (AIRO), and on-line test and compliance tools objects (OTCT). VSMT-OS provides the means by which a logical model can be transformed into a physical implementation using standardised cross-product system components.

VSMT-OS saves you time and money with your product releases earlier and with lower development costs. Risks will be reduced, product reliability and quality increased and maintenance reduced. It has low memory, hardware and processing overheads with machine-speed interrupt response for fastest possible throughput rates. VSMT supports applications programs written in any high-level language.

VSMT is intended to provide one standard methodology covering all producers and users of microprocessor controlled systems. It is possibly the only current technology which takes a product development from marketing conception to customer acceptance.

VSMT provides not only a step by step guide to product development but can also be used as the basis for a company’s transition to BS5750/ISO9000 compliance.

TERMINOLOGY AND CONCEPTS

ASYNCHRONOUS EVENT PROCESSING

This refers to whenever something occurs and an internal queuing mechanism holds the data prior to the relative task being scheduled to process it.

CRITERIA'S

Criteria's can be represented using the logical component descriptor language. This is a method of defining the criteria by which a procedure processes data against requirements and events.

INCOMING MESSAGE QUEUE

Each task has its own unique message queue on which all events pertinent to that task may be queued until the task is able to process it. This structure maintains chronological order of all event stimuli.

INTERRUPT HANDLING

Whenever a hardware event takes place that requires immediate action, the current work being carried out by the CPU is interrupted and not continued until the interrupting event has been processed. When an interrupt first occurs, there is generally a period during which no other interrupts of the same or lower priority can be processed. In order that this period is kept to a minimum so as not to miss other events or a reoccurrence of the same event, interrupt handling is usually split into two areas; the interrupt service routine (ISR) and the base-level interrupt handler (BLIH).

INTERRUPT HANDLING - ISR

When an interrupt first occurs, it's associated ISR is invoked by the CPU and a minimum set of actions performed

The ISR is the part of interrupt handling where other interrupts of the same level or lower are disabled and should be responsible for the high-risk event capture operations only. Once the occurrence of the event has been recorded, either a message is sent to the interested task with no further actions taking place under interrupt or, control can be passed to the BLIH for event processing under interrupt control but with all interrupts enabled.

INTERRUPT HANDLING -BLIH

Whenever event processing is required under interrupt control but without compromising other interrupt activity, a base-level handler is used to process the events in an asynchronous way, relying on the ISR part to guarantee capture of all event occurrences using whatever data queuing structure deemed appropriate to the individual ISR/BLIH protocol.

The base-level handler, once invoked by it's ISR is free to process the event without time critical considerations to worry about. Yes, it must be as fast as is feasible, but the occurrence of other or similar events are able to interrupt the processor, even the base-level itself - thus ensuring nothing is missed.

LOGICAL COMPONENT

A logical component of software is either a task or a utility. Tasks being scheduler driven and utilities being a collection of procedures usually driven by tasks. It always a complete set of related features with few or no dependencies on other components.

MESSAGE

A message is the basic unit for passing information between tasks via operating system primitives.

OPERATING SYSTEM KERNEL

An operating system Kernel is a collection of functions that provides program initiation and data communication services to application programs. It provides an environment within which a collection of application programs may operate in a seemingly concurrent way. Application programs, called tasks, are run by the Kernel in response to requests from other tasks or external events. In the case of an external event, an interrupt handler acknowledges the occurrence and generates a message for processing by the relevant application task. This action constitutes a 'wake-up' or 'scheduling' of the task.

Application tasks may be assigned priorities. The Kernel's scheduler will then choose the highest priority scheduled task to run next.

Application tasks may pass data or 'messages' by calling Kernel primitives. These routines maintain task message queues to store and forward messages between tasks and between interrupt handlers and tasks.

The Kernel must not be confused with operating system. An operating system in the case of VSMT will contain a Kernel but constitutes a much larger provision of system services.

PROCEDURE

A component of software logic driven by a synchronous call/return interface, and under control of the calling software. It consists of a number of process activities performed according to pre-defined criteria's.

PROCESS

A process is a description of a complete area of work and covers all events and actions necessary from start to finish. A process may consist of single or many tasks, and can make use of various utilities. Processes are broken down into tasks and utilities.

RE-ENTRANCY

Usually describes a phenomenon that occurs in shared code segments, but which also occurs in shared data segments.

REAL-TIME SYSTEM

This refers to any system where response time is the key issue whether it be expected in microseconds or comparable to human reaction. This term is used to distinguish such systems from on-line batch or data processing systems in which response times are not the main concern.

SCHEDULER

A scheduler, often referred to as the operating system kernel, is the core piece of code responsible for ensuring all tasks are given a slice of CPU time whenever it has something to do and according to a defined task priority order. 'Something to do' in this environment means whenever at least one message is waiting on a task's incoming message queue or, the task relinquished its slot voluntarily prior to completion in order to give other higher priority tasks a slot, but needs to be continued.

STATE TRANSITION

The transition of a task from a particular stage of a process sequence to another and the definition of all events and action which are appropriate at that stage. In design and implementation terms, a state procedure is the destination of a state transition, from one state procedure to another, and is responsible for receiving and processing messages in that state. Whilst active, a state procedure becomes the schedulable task in the operating system's scheduling environment.

SYNCHRONOUS EVENT PROCESSING

This refers to whenever something occurs and the event is processed immediately by the currently executing task. No descheduling occurs between the event occurring and being processed.

SYSTEM

A system is the complete embodiment of a hardware and software solution which fully satisfies the product requirements. A system is broken down into processes.

TASKS

A task is the collective name given to a set of software modules which are scheduled to execute in its own 'virtual' processor environment. Tasks are uniquely identified and assigned priorities. The Kernel's scheduler will always choose the highest priority scheduled task to run next.

Application tasks may pass data or 'messages' by calling Kernel primitives. These routines maintain task message queues to store and forward messages between tasks and between interrupt handlers and tasks.

UTILITY

A collection of logically bound procedures. They may be defined either for exclusive use by a process or task or as an unrestricted system-wide support service.

VIRTUAL STATE MACHINE

A component piece of an application task which executes at a given time in a process sequence with a remit to only respond to pertinent events at that time.

VOLUNTARY SUSPENSION

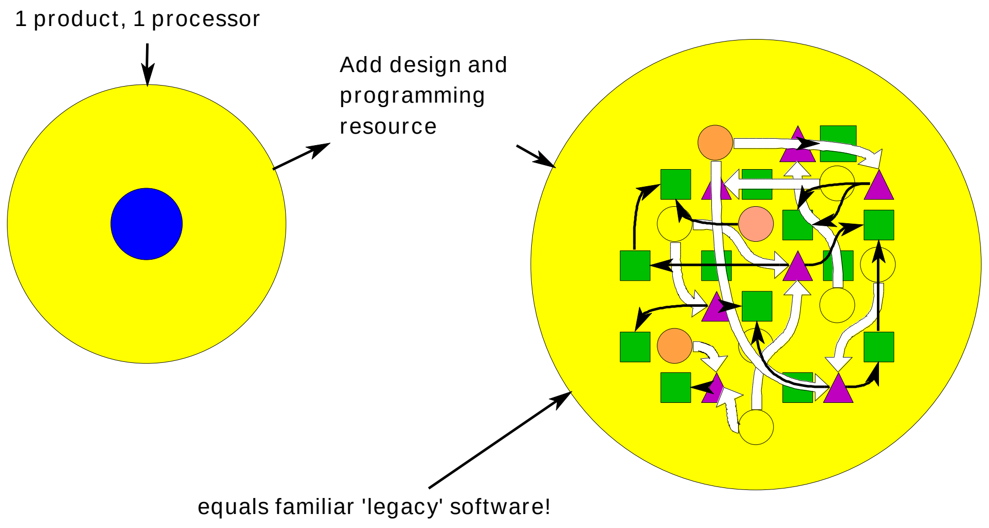
Also known as co-operative co-existence, this refers to when a task relinquishes its slot voluntarily in order to allow other higher priority tasks to be scheduled.

WHY VSMT-OS?

Designing and implementing applications using VSMT is not only simple but standardised for every system you're likely to develop. The next few pages describe the major steps to be taken when designing a system and provides the basic structures for all task component source code.

THE 'REALITY'

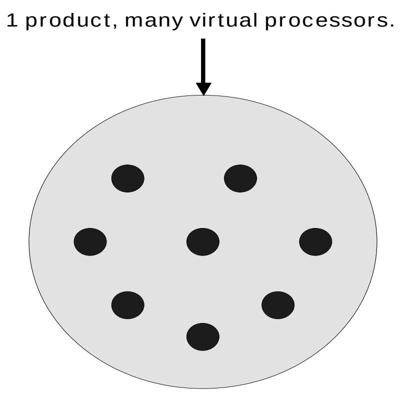
Invariably, you will have one or more products to develop, each with at least one embedded microprocessor. Often, a poor design is followed by a poor implementation and the resulting embedded software is a 'headache' to build and a 'nightmare' to test and debug.


1 product, 1 processor
Add design and programming resource
equals familiar 'legacy' software!

However, by following the basis of Virtual State Machine Technolgy and it's associated methodology, the transition from conception to implementation is made much simpler. The first major step is the partitioning of system functionality into individual processes.

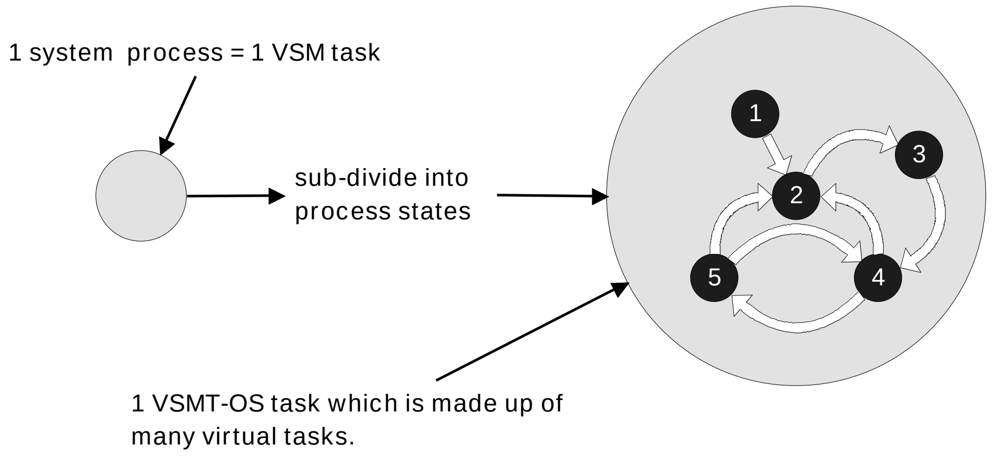
THE 'DREAM'

Every process defined in a given system is simply defined to be either a set of library procedures or a VSMT-OS schedulable task. For those to be a schedulable entity, they are named virtual processes.



VSMT-OS TASKS

Once you have specified your system processes (virtual processors), you have the basis for the definition of a VSMT-OS task. Each task is simply divided into separate states of operation which reflect the current state of operation at a given time. Each of these virtual states become the schedulable item in it's task environment when the particular state is active. This means that any given task is capable of supporting multiple virtual tasks within its own environment.

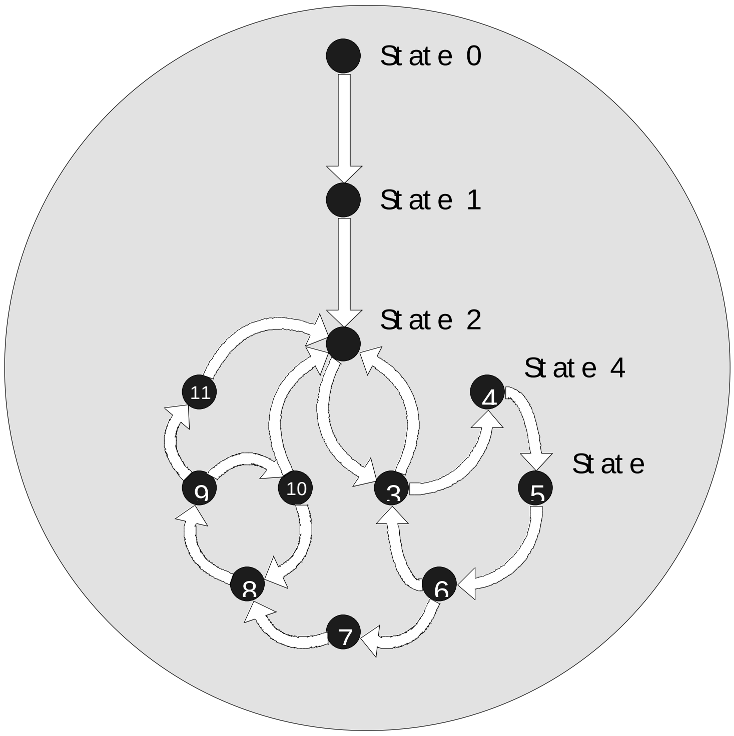

1 system process = 1 VSM task
sub-divide into process states
1
2
3
4
5
1 VSMT-OS task which is made up of many virtual tasks.

Each virtual task is a schedulable task operating mutually exclusively while active, in the domain of the parent process, co-existing with other virtual tasks in the same VSMT-OS task environment. This in its entirety, is one Virtual State Machine. Since other processes in the system will have their own VSMT-OS tasks defined, a system will typically be made up of many Virtual State Machines.

VSMT TASK MODEL

Every Virtual State Machine should conform to the following model for its design and in this way, is both flexible to meet the needs of each system and easily understandable and recognisable amongst project members. Although a Virtual State Machine may have as many states as required, the first three state are defined to be consistent in the fact that they always provide the same areas of functionality.

State 0 cold start	- 'power up'
State 1 warm start	- 'task restart'.
State 2 idle	- 'awaiting requests'.
States 3 to 'n'	- application specific.



State 0 - ‘cold start’

State 0 is always the power-up initialisation procedure and is often referred to as ‘cold start’. It is a standard call/return procedure, does not rely on any prior initialisation of VSMT-OS, and does not make any use of VSMT-OS system services. Primarily, since it is invoked immediately after power being applied to the equipment, it should quickly ‘silence’ all task related hardware leaving it in a known, quiescent state in readiness for normal operation. State 0 is invoked by the VSMT-OS start-up files with the precise procedure name being specified in the static start-up data.

Every state 0 procedure should look similar to the following structure.

```
PROCEDURE STATE0_cold_start

⇨ Initialise task related hardware.
⇨ Initialise task related software data.
⇨ Initialise anything else.
RETURN

END STATE0_cold_start
```

State 1 - 'warm start'

State 1 is always the task restart procedure, is often referred to as 'warm start', and typically follows an operator request for specific task related re-configuration or a level of failure in the system is deemed sufficient for the restarting of individual tasks. State 1 procedures may be invoked selectively either by a system service or by the task itself, so it is system dependant as to how many and which tasks restart at a given time. Unlike state 0, it is not necessarily a standard call/return procedure as it may be and is generally designed according to the conventions for a state procedure. That is, it does make use of VSMT-OS system services, usually in the form of waiting for the correct configuration or 'ready to operate' messages. Again, it can be just a call/return procedure but in principal, is responsible for re-initialising / configuring its task data environment prior to returning to normal operation. It must however, complete its actions with an explicit transition to state 2. State 1 is the first entry point into the specific VSMT-OS Virtual State Machine task environment and is invoked by the scheduler following power-up. The precise name is again specified in the static start-up data.

Every state 1 procedure should generally look similar to the following structure.

```
PROCEDURE  STATE1_warm_start

DO FOREVER

  ⇨ Wait for next message
  ⇨ Read message

  SWITCH ON message type

  ⇨ Initialisation message
    ⇨ Perform related actions.

  ⇨ Configuration message
    ⇨ Perform related actions.
    ⇨ Release message buffer.
    ⇨ Make transition to 'state 2 idle'.

  END SWITCH
  ⇨ Release message buffer

END DO

END STATE1_warm_start
```

State 2 - ‘idle’

State 2 is always the task idle state. The state in which the task has successfully initialised and is waiting for requests to commence processing operational features. Following every operational request, feature processing may take the task through a number of it’s VSMs and, irrespective of whether the feature was successful or not, a return to state 2 at the end of feature processing is mandatory. This means that whenever a task is doing noting, i.e. waiting for the next request, it is in a standard, known state. Obviously, this and subsequent states can make use of the full set of VSMT-OS system services.

Every state 2 procedure should generally look similar to the following structure.

```
PROCEDURE  STATE2_idle

DO FOREVER

   ⇨ Wait for next message
   ⇨ Read message
   SWITCH ON message type

   ⇨ message x
      ⇨ Perform related actions.
      ⇨ Release message buffer.
      ⇨ Make transition to new state.

   ⇨ message y
      ⇨ Perform related actions.
      ⇨ Release message buffer.
      ⇨ Make transition to new state.

   ⇨ other messages
      ⇨ Perform related actions.

   END SWITCH
   ⇨ Release message buffer

END DO

END STATE2_idle
```

State ‘n’ - all others

All other state procedures are completely application specific, both in their numbers and their character. Essentially, they must at least wait on messages, process them and perhaps make state transitions. There is though no reason why a VSMT-OS task consists of no more than states 0, 1, 2 and 3 with state 3 being the entirety of the operational side of the application. This is generally unlikely as the application itself would be unusually simple and quite probably not making use of the Virtual State Machine philosophy.

All state ‘n’ procedures should generally look the same as the example given for state 2.

OVERVIEW

The use of a real-time, multi-tasking executive is necessary in so much that the physical model, ie. the implementation, can be created from a logical description of a system according to the architectural concepts promoted by VSMT.

A real-time, multi-tasking operating system must provide software designers with the mechanisms to build event driven systems whilst not compromising the need to implement logically partitioned components of software. It must aid both re-use and portability of logic, along with reduced development and maintenance costs. With a small yet rich set of features, all the tools necessary to monitor, control and command concurrent processes will be provided, based on well-known, simple and effective software mechanisms. Both design and implementation of product specific applications software can be achieved with the majority effort and focus kept on the specification of tasks and their supporting state machines. These state machines should then precisely reflect the high level sequence of events and actions involved with each operational process.

VSMT-OS permits each major functional process in a real-time system to be designed and coded as a separate task program. Tasks run concurrently, sharing the processor and other hardware. In addition, each task is assigned a priority to ensure that important functions take precedence over less important ones.

VSMT-OS consists of the following architectural features;

- Configurable number of virtual state machines, each comprising unlimited numbers of virtual tasks, individually scheduled as concurrent processes.
- Integral state transition handling to support task state transition design architectures - no more global state variables or preceding scenario analyis.
- Dynamic task allocation/termination, with multiple copies supported.
- Non pre-emptive, reduced context saving task scheduling environment.
- 'Run until current event processed' task execution
- Pre-emptive interrupt scheduling.
- Base-level interrupt handling for guaranteed capture of i/o events.
- Inter-task message communication.
- Memory management and real-time event and status timing.
- Start/stop time-out handling and time/date event handling.
- Low level, high resolution hardware scanning.
- User configurable, any language, any processor, any host, any target; compile & go!

The run-time features, described above, are satisfied by a set of system components known as Application Independent Run-time Objects (AIRO) and will always be embedded in the target system.

To support testing, an additional set of components, known as the On-line Test and Compliance Tools (OTCT), are provided and co-exist in the target system during integration and acceptance testing only.

MECHANISMS

This section describes the mechanisms, by which the needs of software designed for concurrent software architectures in real-time embedded environments, are met.

THE KERNEL

AIRO-KE consists of three major components; a task scheduler, a set of message passing functions, and task control functions. The scheduler initiates tasks on a priority basis in response to messages sent from interrupt handlers and other tasks. Messages are passed between tasks and between interrupt handlers and tasks by means of linked list message queues, and the monitoring and high level control and coordination of time in relationship to task activities is supported via timing queues and real-time clock maintenance.

AIRO-KE is a non pre-emptive scheduler where no task is able to interrupt one already running, even if of a higher priority. The exception to this is the base-level interrupt handling which supports full i/o pre-emption on the task executing. Each task however, must complete current processing and relinquish its processor slot itself. Except for interrupt servicing, the running task has control of the processor until it suspends itself. In such scheduling systems, it is necessary for tasks to co-operate in using processor bandwidth by limiting the amount of processing performed between voluntary suspensions. This is not a limitation, rather an advantage since it considerably simplifies the scheduler design and therefore reduces memory and execution-time requirements. Another major advantage of non pre-emptive scheduling is that it reduces the possibility of inconsistent data being passed or shared between program components. It can always be assumed that a task will complete the compilation and reception of entire messages before voluntarily suspending itself. With a pre-emptive scheduler, it can never be known up to what point a task has executed prior to being pre-empted.

It must be made very clear that all event stimuli in the system represent themselves to the interested task in the form of a message. It is the generation of a message by a third party system component which causes the scheduler to prepare a message for task message reception and subsequently schedule the appropriate task at the next available slot.

INTER-PROCESS COMMUNICATION

Each task has a single incoming message queue configured for use like a personal in-tray. In order to receive information, a task must wait for co-operating parties in defined dialogues to supply information via it's incoming message queue, as and when events occur. Tasks are only scheduled when at least one message is waiting on it's queue. Depending on the priority and load of other tasks, a task might have many messages waiting to be serviced. Since only one piece of code can occupy the CPU at any given point in time, all messages created are guaranteed to be queued in precise chronological order on an incoming message queue.

It is possible to implement a facility which effectively re-instates the last message read from the incoming message queue. Usually, the wait_for_next_msg primitive copies the message data queued in a buffer to the static area in the task specific data. This is in effect a destructive read and the next time the primitive is used the area is rewritten with the next message. However, this primitive would set a flag so that the next time a wait for next msg primitive is called, no next message copying occurs. The last message is still in the task data.

TASK LIFE-CYCLE

All task types must be defined and given a priority at compile time. This information is part of the static configuration data and requires the specification of a complete operating environment in advance. This is only requesting the types of task which are to operate and not the number of tasks of a certain type. After power-up, tasks may be created and terminated dynamically during normal operation, provided that the task type was defined in the static data. Consequently, varying numbers of the same task type may be active on the scheduler list at a given time.

Tasks may therefore be active for as long as the equipment is switched on or for indeterminate periods relating directly to the occurrence and duration of process sequences. During the entire operation of a system, a task can assume one of three states at any time;

- Nothing to do so the task is ***waiting for a received message***.
- As a result of a message being queued on a task's incoming message queue there is now something to do and the task is therefore ***ready to be scheduled*** at the next available slot.
- There is a message to be processed and the task, having been given a scheduler slot is now ***running***, exclusively holding the CPU's execution thread.

PROCESS STATE HANDLING

Every process must be able to realise a need to change its expectations of it's environment as it progresses through operational activities. A state transition environment is therefore needed to allow an operational, situation specific structure without the need to remember past events using for example, global variables. State procedures for process state and dialogue handling provides histories of events and actions, inherently in the design eliminating the need for confusing and often abstract supporting data.

Every task is a schedulable entity with a number of state machines ready for action at the appropriate time. From the start to finish of a process, transitions are made from one state machine to another yet at any given time, only one state machine in a task is ever active, and whilst active is the schedulable 'task'.

All task copies move through a sequence of states which are in direct relationship to the high level dialogue being processed - whether that be primarily driven by hardware or software stimuli. The following paragraphs describe how the state transition principles need to be implemented in code.

- Every state, which should be shown in a state transition diagram, is represented by it's own state procedure.

- The processing of a message in a given state is referred to as task execution. There may be a variety of messages expected in a given state and task execution may not necessarily result in a state transition.

- A state transition can be effected simply by the user calling any of its state procedures. This allows a task to change its environment to that specified by the new state. The new state will then become the next 'virtual task' in the machine.

- In order to support dynamic transitions from one state to another at any point during task execution, the 'wait for next message' primitive must be invoked on entry to the new state procedure. Not only does this deschedule the task but it sets the stack pointer to an initial value - thereby destroying any return addresses from prior task execution procedure call nesting.

- Each state reflects the current stage at which a process is being dealt with and explicitly reflects all events expected at that stage along with any associated timing constraints.

UNIQUE TASK IDENTIFICATION

Every schedulable task has a unique identity called a task instance. Each task instance consists of a task type and a task copy number. This allows multiple copies of a given task in order that many resources of the same type can be handled separately. For example, if there are up to twenty transmission lines having the same characteristics, then twenty copies of a 'tx task' can be created dynamically when required, each subsequently executing as individual tasks. Obviously, with dynamic creation and termination of tasks, management software must be designed to co-ordinate and control these activities for each set of resources and processes requiring multiple task copies. In addition, multiple task copies generally require more high-level process dialogues to be specified during system design.

SCHEDULING CRITERIA

Scheduling is based on a round-robin, priority ordered algorithm which means that the highest priority task currently ready is given the next execution slot once the presently running task finishes. A task assumes its priority by it's position in the scheduler list; top of the list has highest priority, bottom of the list has lowest.

TASK RE-ENTRANCY FOR MULTIPLE COPIES

In order to support multiples copies of a particular task, each task is allocated it's own task data area. This is linked to individual task descriptor records and is under the control of the design of each task as to it's structure and usage. In the configuration data for all tasks, a reference is set to the task variable for each task type which is used to store the current task pointer. During start-up and prior to every task execution i.e. each scheduler slot, the task data associated with the task about to be scheduled is set up by the kernel and the task is then able to access task specific data in a re-entrant way as if the task was a single copy. This does mean however that each individual task's source code must declare a global pointer according to a defined convention and the OS configuration data set with a reference to that global.

EXTERNAL WORLD PROCESSING

The external world is the source of all events, excluding internal integrity checking, and as such may be split into the following event categories.

- Something has happened (state change or occurrence indication). This should result in at least a process action.

- Something expected has happened within a given period of time. This should result in a time-out being stopped and a process action.

- Something expected hasn't happened within a given a period of time. This usually means a time-out has expired and a process action needs to be performed.

ASYNCHRONOUS EVENT HANDLING AND COMMUNICATION

All events in a system are effectively occurring asynchronously to everything else but due to the singular execution nature of a processor, events need to be integrated into the application environment for apparent sequential processing. By converting events into messages which can be buffered in time occurrence order on respective task queues, a mechanism is achieved whereby through selective scheduling by the scheduler, all events are 'spun' into a single thread and seen to be processed in a sequential manner. Different tasks performing different activities only when required. This means every task has a single communications access point, the incoming message queue. Although utilities are encouraged to provide low-level control of the physical interface with the world, communication between tasks must always conform to the protocol of message passing. Dialogues between tasks and processes can then be established according to the high-level requirements of the system.

TIMED EVENT MONITORING

Requests for timing of events are integrated into the OS timing environment and on expiry of timers, message created and queued on the requesting task's incoming message queue. If a task has nothing to do whilst waiting for the expiry of a timer, then it is never scheduled, and remains *waiting* until the time-out message is queued and the next available slot given.

Every concurrent timing request requires its own individual timing buffer so if a task requires three concurrent and two non-concurrent timing requests, the task needs to allocate four of it's own timing buffers. In addition to the message driven real-time monitoring, it is possible to utilise an alternative method to provide extremely accurate and high resolution monitoring of external hardware or events. Usage of this must be kept to a minimum and only according to specific rules since it makes use of the interval timing interrupt environment. For any task requiring this feature, a reference to a scanner routine, produced as part of the task design, needs to be registered into the OS environment either during initialisation or as part of static task descriptor configuration data.

INTERRUPT HANDLING

It is the ISR and it’s base-level part that provides the low level intelligence necessary to minimise the number of messages generated. However tracking or notification of key events captured whilst under interrupt control should be communicated to the appropriate task using messages, in order that high-level analysis and interpretation may be performed by the task.

CODE RE-ENTRANCY

Whenever a piece of code can be called by both task and interrupt software, it is vulnerable to data contention. The code must be made resilient to this when it is running under task execution and then interrupted and called again within the interrupt handler prior to the initial call being completed.

This can be achieved by ensuring that all working data is either passed via the stack in the procedure call, stored on the stack via local variable declarations with compilers which allocate local variables on the stack frame, or allocate memory every time it is called. Although this will overcome data contention, it will not guard against the possible corruption of event sequence. To do this, a semaphore or resource 'lock-out' mechanism must be applied either in software or in hardware by disabling associated interrupts until the code has completed.

DATA RE-ENTRANCY

Whenever a system has to process events from many peripheral devices or resources of a similar type, a single piece of code should be designed to operate in a data re-entrant way. This means that a single piece of code is executed to process all events from all devices of the same type, rather than have a separate piece of code for each device. However, for this to work with no data contention, each device must have a data area allocated to it to store operational, control and configuration data. This must be performed at system initialisation or as soon as an individual device becomes 'on-line'. In order for the data re-entrant code to operate on the correct data set, the identity of a device must subsequently always accompany any event data when the code is called.

BUFFER HANDLING

To minimise the need to foresee system resource requirements in advance, a combination of tasks providing their own resources and maximising the availability of shared facilities using resource managers is pursued wherever possible. For example, timing buffers are supplied by individual tasks who require them, and message buffers are supplied by the kernel from a shared pool but never accessed by a task.

ENVIRONMENT INITIALISATION

The Kernel is the effective start-up code following hardware initialisation and performs first-time task scheduling in order that tasks are set to a known, initial state. Two stages of initialisation are recommended before a transition to the 'idle' state.

- State 0, hardware reset, cold / power up restart. Initial 'gagging' of noisy, volatile hardware.
- State 1, software task restart, warm restart, task general SW / HW initialisation.
- State 2, idle state, state specific SW / HW initialisation

OS CONFIGURATION

The only requirement for static, compile time data is to enable the kernel to initialise the scheduling environment and 'kick off' all tasks. This is satisfied with a list of task descriptor records and an environment descriptor record, which needs to be created for each and every product. This will soon be taken care of graphically using the VSMT-OS windowsTM based configuration tool which is to be released in the near future.

'WATCHDOG' CONTROL

If a system requires the ability to check whether a task 'lock-up' has occurred, i.e. a task has entered an infinite loop, then the designer must specify a 'watchdog' mechanism.

The first method is to specify a task to work in conjunction with both a hardware reset counter and a separate interval timer. This task is configured as the lowest priority task and continually waits on a repeated interval time-out expiry message. There are two levels of watchdog monitoring. The first which relies on the watchdog task being scheduled a faster frequency to that of the hardware reset counter. When scheduled, the counter is reset and thereby stopping a hardware reset of the entire system taking place. This method does not rely on the timing utilities interval timer interrupt.

The second method provides the capability to monitor for task 'lockup' and perform individual task restarts on detection. If the any task causes itself to be restarted more than a given number of times, then the overall hardware reset counter is allowed to override since an individual task may not be restarted more than 'n' times. This method relies on both the hardware reset counter and the timing utility interval timer interrupt periodically checking for 'stuck' tasks. This relies on the kernel incrementing a modulo 64k counter prior to every time it launches a task and the interval timer checking for task reschedules over a period of time. The watchdog task still resets the hardware timer as normal.

The advantage of the second method is that it improves the apparent reliability of equipment by allowing the restarting of a 'bugged' task rather than restarting the complete system and interrupting services for what may be a transient, infrequent faults. From a debug point of view, it also provides a mechanism by which the conditions surrounding a fault can be logged and also to collect scheduling statistics for individual task occupancy analysis.

Another method is to implement a mechanism which relies on interaction between the timer interrupt procedure and scheduler, therefore eliminating the need for an additional hardware timer/counter.

DEPENDENCIES

PROCESSOR OCCUPANCY

For any given processor environment, each task should not hold the processor for more than about 5ms in any given scheduler slot. Of course this is only a rough guide and to make estimation in real terms easier, a translation of the typical execution time of the main language constructs should be made. Under the 'performance' section, a rough guide to how long typical activities take is given and is a fairly good basis for specifying system partitioning and work-load splits. This will enable individual engineers to make fairly accurate correlation between the number of executable language statements and the time required for all execution threads.

PRIORITISATION

The greater the urgency of service to a task when an event occurs, the higher on the scheduler list it should be.

TASK DESIGN ARCHITECTURE

All designs must embody an event driven architecture with a sequential machine approach. All system features and activities are a sequential set of events. Concurrency only allows a system to process multiple activities occurring in parallel.

ZERO-WAIT CONFORMANCE

Never wait using task internal timing loops. Event driven systems dictate that you never 'pace around' searching for something to happen - you define exactly what you are required to deal with, 'snooze' until something related happens, deal with it and go back to sleep. Ignore everything else. This is the only way in which maximum utilisation of processor capacity can be achieved. It also forces a simpler design.

VSM PRIMITIVE UTILISATION

In the interest of coherent design, readability, and ease of understanding, it is recommended that all operating system primitives are kept within the highest and lowest level procedures in the software hierarchy. This will maximise the opportunity for the re-use of general purpose utility software.

SELF-INVOCATION

A task should never usually need to send a message to itself. Self invocation is generally time related and provided by requesting timing messages. However, in exceptional circumstances, sending a message to itself can be used as a way of forcing a fast reschedule.

MINIMAL 3-STATE AWARENESS

Task design must always follow the minimum 3 states model.

SPONTANEOUS TASK INVOCATION

Parameters should never be passed to a state procedure, nor should a state procedure define its own local variables. Any task data must be defined as global to the task module, outside the scope of the state procedure itself. Failure to follow this will result in software execution errors and unreliable task operation.

DESIGN CONSIDERATION

TASK DESIGN CONSIDERATIONS

- Split system into small function blocks to become tasks.
- Tasks should then be sub-divided into states.
- State 0 is the power-up initialisation (cold-start) code for the task.
- State 1 is the warm-start (software restart) code for the task.
- No messages to be sent from state 0s.
- No state transitions in state 0.
- Minimise task global data. No global data to be shared between tasks.
- Keep inter-task message communication to a minimum. This is not a limitation of the OS but advice to simplify design and a maximise system performance. Why use up valuable cpu time passing messages when for example a call/return to a library function might be better.
- No task should hold the CPU for more than approx. 5ms in any one scheduler slot, aim for less.

VSMT-OS is the generic name for 21-dC's family of real-time, multi-tasking operating systems, and is an integral part of its Virtual State Machine Technology. VSMT-OS is available for most microprocessors available today, with most of the manufacturers processors catered for whether it be 8, 16 or 32-bit architecture. The source code is all in ANSI 'C' so is portable from one processor to another and target cpu specific code has been reduced to just a few lines of assembler. These few lines of assembler are the only part of the entire OS needing to be changed for each specific processor and is easily achieved by in a few minutes by someone knowing the basic target cpu instruction set. This means that one operating system implementation shares the same design philosophy so once familiar with the features and architectural mechanisms of the generic VSMT-OS, the designer and implemented can comfortably and effectively switch target processor environments with great ease. This allows the effectiveness of both engineering staff and current hardware solutions to be utilised to their full, using any of the mainstream programming tools.

The success of a product operating in a real-time environment is gauged not only by the speed by which it can respond to real-time events, but also by the number of such events it can effectively process at times of maximum activity. VSMT-OS reduces response time to a minimum, provides inherent buffering of captured data, and ensures maximum throughput at peak times. VSMT-OS uses multi-tasking, interrupt to event message signalling translation, priority task scheduling and pre-emptive interrupt scheduling.

Multi-tasking allows the CPU processing capacity to be shared between all system tasks in such a way that the CPU is only idle when none of the tasks have anything to do. Whilst one task waits for an external event or the passage of time, other tasks with immediate processing needs is given the execution thread.

Priority is based on the order of tasks in the scheduler list. First task in the list assumes highest priority.

In most applications, development of the systems software takes at least 20% and often 50% of total development time. Development projects using VSMT not only benefit from a saving of up to 50% of effort but, benefit even more by the structured design and implementation methods promoted. Also, during test integration, integral test and performance tools facilitate rapid functional system debugging.

A large number of messages being passed between tasks may indicate that the decomposition criteria for splitting the system into tasks is flawed. Since message passing increases inter-task dependencies, the interfaces to the features offered by tasks must be clearly thought through and carefully defined at the design stage.

Try to ensure that VSMT-OS system calls are kept either at the highest or lowest level of the task code hierarchy in order to maximise portability of logic and to significantly ease the test and debug phase.

Each internal task has its own task descriptor record containing local data, run time data and information relating to the control of that task running under the OS. A task record exists for the lifetime of a task. A task itself can be dynamically created and disposed of.

DATA HANDLING

All RAM which is not used specifically by tasks or VSMT-OS are placed in the memory managed environment and is available to individual tasks on a request basis. In this way, each task only reserves the amount of memory needed at a given time rather than the maximum it would require under peak loading.

Allocation of memory blocks of varying sizes is possible. Efficient allocation. Fast liberation. No garbage collection necessary due to immediate rebuilding alongside neighbouring free blocks.

BUILD CONSIDERATIONS

VSMT-OS requires both RAM and ROM resident data. Since these two data types are segregated, the users system can specify VSMT-OS and the application code to be ROM resident and thereby eliminate the need for bootstrap code of separate OS ROM chip.

FOREGROUND ACTIVITIES

A foreground task is a priority activity continually scheduled by the OS and is therefore able to respond and control events occurring in the world around. Foreground events correspond to the reception of messages, the expiry of timed periods, pre-emptive i/o events and will not be interrupted whilst only lower priority work is outstanding.

BACKGROUND ACTIVITIES

A background task is given a defined period of time to operate when no foreground activities are outstanding. Background tasks are the lowest priority in the system.

REAL-TIME CONTROL

VSMT views time from both the human and product point of view and therefore supports time and date along with absolute system time. This gives any process the ability to associate events with the real world time of daily life. Additionally, absolute tagging of events to the internal world of the virtual machine is possible so as to provide unique logging of events as a once only moment in time. This gives unique reference data for time chronological sequence analysis.

APPLICATION INDEPENDENT RUN-TIME OBJECTS (AIRO)

For a detailed guide to all Application Independent Run-time Objects (AIROs), refer to the User Interface specification: **uiairo.doc**

VSMT-OS is the generic term for the complete package of VSMT-AIROs. All VSMT-OS user interfaces are detailed in the document 'ihVSMTos'.

VSMT-OS/AIRO consists of various functional areas, each covering a specific aspect of the operating system. These areas are;

- **Kernel**
- **Real-time**
- **Memory Management**
- **Queue Management**
- **General Utilities**
- **System Security**

KERNEL PRIMITIVES AIRO-KE

These provide the tools for inter-task message passing and task scheduling.

- **Get task identification**, provides a task with its unique identification, differentiating it from all other tasks.
- **Activate task,** allows a task to be created and configured into the active scheduling environment.
- **Terminate task,** allows a task to remove a task from the active scheduling environment.
- **Reserve message buffer,** provides a ‘free’ message buffer for the compilation of a message prior to inter-task communication.
- **Send message to**, allows a task to send a message to another task. This does not result in the sending task being descheduled. This can also be used in interrupt service routines to notify task software of external events occurring.
- **Wait for next message**, forces immediate descheduling of the task and allows higher priority task to execute. The descheduled task is left in either a ‘*ready to run at the next available slot*’ or ‘*waiting for a new message to be received*’ state.
- **Receive message**, allows a task to read the next message on its incoming message queue.
- **Re-read message**, allows a task to restore its reference to the last message read.
- **Reschedule,** allows a task to descedule itself but resume immediately once any higher priority tasks have executed. NOT OFFERED AS A FEATURE SINCE THIS REQUIRES TASK SWITCHING WITH CONTEXT SAVING OF REGISTERS. Effective 'reschedules' take place whenever a state transition or **Receive message** is invoked.
- **Delay,** allows a task to deschedule itself and not resume until a specified period of time has elapsed. NOT OFFERED AS A FEATURE SINCE THIS REQUIRES TASK SWITCHING WITH CONTEXT SAVING OF REGISTERS. Can be implemented by using AIRO-RT:**Start time-out**.
- **Release message buffer,** returns a ‘used’ message buffer back to the ‘free’ message buffer pool.

For detailed descriptions of the associated logic, refer to the Logical Component document: **lcairike.doc**.

REAL-TIME PRIMITIVES AIRO-RT

These provide the tools for real-time event and process timing.

- **Initialise timing buffer**, prior to using any of the following real-time primitives, each 'task declared' timing buffer must be initialised.
- **Start time-out**, requests that a time expiry message is sent to the task after a specified duration. This is the facility to use for high frequency event timing.
- **Stop time-out**, requests that an outstanding time expiry message, i.e. one that is currently active, is immediately cancelled.
- **Request a wake-up**, requests that a wake-up message is sent to the task at a specified time and date.
- **Request a regular prompt**, requests that a prompt message is repeatedly sent to the task at a given time of day, every day.
- **Request reminder**, provides a reminder message on the requesting task's incoming message queue in a specified time from the request.
- **Cancel request**, allows any of the previous requests to be cancelled.
- **Get system time and date**, supplies the current system date and time of day.
- **Set system time and date,** allows the current system time and date to be updated.
- **Get absolute time**, supplies the absolute system time since powering up.
- **Get time-stamp,** provides a unique time reference for tagging events in chronological order.
- **Check if summer time,** shows whether the current time is local Mean Time or local Summer Time.
- **Set local summer times**, allows the start and end, date and times of Summer Time to be entered.
- **Check date not old**, verifies whether a given date has passed or not.
- **Check days in month,** checks that a given number of days in a month are correct.
- **Validate time and date,** allows a given time and date to be validated.
- **Compare time dates,** allows a comparison to be made between two given times and dates in order to see if one is earlier, the same or later.
- **Add to time,** allows a time and date to be extended by a specified time and date duration.

MEMORY MANAGEMENT PRIMITIVES AIRO-MM

These provide the tools for buffer resource handling.

- **Allocate buffer**, supplies a dynamically allocated buffer of any size from system memory resources.
- **Deallocate buffer**, returns a previously allocated buffer to the system memory resource.

QUEUE MANAGEMENT AIRO-QM

Queue Management Primitives (Linked Lists)

These provide the tools for manipulating linked lists.

- **Initialise linked list**, prior to use, a link list must be registered during task initialisation.
- **Append to top**, appends a record to the top of a linked list.
- **Append to bottom**, appends a record to the bottom of a linked list.
- **Append before last read**, inserts a record into a linked list at the position before the last record read whilst performing relative list accesses.
- **Read element from top**, read but don't remove the record at the top of the linked list.
- **Read element from bottom**, read but don't remove the record at the bottom of the linked list.
- **Read next element in list**, read but don't remove the next record in the linked list following a previous relative access.
- **Remove element from list**, remove a specified record from a linked list.
- **Remove element from top**, read and remove the record at the top of a linked list.
- **Remove element from bottom**, read and remove the record at the bottom of a linked list.

Queue Management Primitives (Cyclic Buffers)

These provide the tools for manipulating cyclic buffers.

- **Initialise cyclic buffer**, prior to use, a cyclic buffer must be registered during task initialisation.
- **Add to head**, add data element to the head of the buffer to become the 'oldest'.
- **Add to tail**, add a data element to the tail of the buffer to become the 'newest'.
- **Read from head**, read but don't remove the 'oldest' entry.
- **Read from tail**, read but don't remove the 'newest' entry.
- **Remove from head**, remove the 'oldest' entry.
- **Remove from tail**, remove the 'newest' entry.

SCANNER PRIMITIVES AIRO-SC

These provide the tools for low-level, high resolution hardware scanning and control.

- **Register scanner**, allows a scanner to be configured into the scanning environment.
- **Commence scanning**, instructs that a specified scanner should now start.
- **Finish scanning**, instructs that a specified scanner should now stop.

GENERAL UTILITY PRIMITIVES AIRO-GU

These provide the tools for general data manipulation.

- **Time to ascii**, converts binary time to its ASCII representation.
- **Ascii to time**, converts an ASCII time string to its binary form.
- **Integer to ascii**, converts a binary integer to its ASCII representation in either decimal, hexadecimal, or binary.
- **Ascii to integer**, converts an ASCII string representing either a decimal, hexadecimal, or binary value to its binary integer equivalent.
- **Binary to BCD**, converts a binary value to its BCD form.
- **BCD to binary**, converts a BCD value to its binary form.
- **Ascii to BCD**, converts an ascii string to its BCD form.
- **BCD to ascii**, converts a BCD value to its ASCII form

SYSTEM SECURITY PRIMITIVES AIRO-SS

These provide the tools for system and data integrity.

- **Calculate checksum**, calculate a checksum for a given area of ROM.
- **Verify checksum**, verify a checksum against an area of ROM.
- **Calculate CRC**, calculates an 8-bit CRC for a specified data set.
- **Verify CRC**, verify a CRC against a data set.
- **Perform ROM test**, diagnoses a section of ROM for faults.
- **Perform RAM test**, diagnoses a section of RAM for faults leaving it either unchanged or initialised to a set value.

ON-LINE TEST AND COMPLIANCE TOOLS (OTCT)

For a detailed guide to all On-line Test and Compliance Tools (OTCT), refer to the User Interface specification: **uiotct.doc**

Prior to final system release, the OTCTs are removed from the build specification and the test overhead released. For optimum results, it is recommended that the various facilities are used in conjunction with each other. The features offered by the OTCTs are specified now in overview.

VSMT-OS/OTCT consists of various functional areas, each covering a specific aspect of the test environment. These areas are;

- **Event and Sequence Tracking**
- **System Monitoring and Performance**
- **Trace and Fault Creation**

EVENT AND SEQUENCE TRACKING PRIMITIVES OTCT-ET

These provide the tools for tracing the occurrence and sequence of inter-task messages.

- **Set surveillance details,** allows specified types of messages to be defined as those to be monitored and logged.
- **Commence surveillance**, using the surveillance details, monitor and log messages.
- **Cease surveillance**, stop monitoring and logging messages.
- **Report surveillance data,** produce a surveillance data set reporting all pertinent messages along with event and sequence data.

SYSTEM MONITORING AND PERFORMANCE PRIMITIVES OTCT-MP

These provide the tools for the collection of real-time system performance data.

- **Start task monitor**, commence monitoring the tasks CPU occupancy.
- **Stop task monitor**, stop monitoring the tasks CPU occupancy.
- **Get occupancy data**, supply task occupancy statistics.
- **Start system monitor**, commence monitoring the entire system.
- **Stop system monitor**, stop monitoring the entire system.
- **Get system statistics**, supply system performance and activity data.

TRACE AND FAULT CREATION PRIMITIVES OTCT-TF

These provide the tools for tracing operational data and creating fault conditions.

- **Trace data**, allows areas of code and data to be logged when executed.
- **Set fault criteria**, allows the criteria for fault insertion to be defined.
- **Commence bugging**, commence tracing the specified data and insert faults when appropriate.
- **Cease bugging**, stop tracing and inserting faults.
- **Report bugging results**, provides a data set reporting all traced data and the sequence of faults inserted.

PERFORMANCE

150micro-seconds for complete message interchange. get a buffer, compile message, send message, deschedule, schedule recipient task.

400,000 message transactions per minute.
6,666 per second

STATE EVENT HANDLING IN A MEASURE OF TIME

150 microseconds for message generation and OS reschedule.
850 microseconds for message event processing
requiring a total of 1 state event transactions per millisecond.

This gives a total state/event processing rate of 1000 per second.

STATE EVENT HANDLING AS A MEASURE OF SOURCE CODE STATEMENTS

7 msgs/second with each message executing 30,000 simple C statements.
21 msgs/second with each message executing 10,000 simple C statements.
226 msgs/second with each message executing 1000 simple C statements.
386 msgs/second with each message executing 500 simple C statements.
717 msgs/second with each message executing 250 simple C statements.
1478 msgs/second with each message executing 100 simple C statements.
2288 msgs/second with each message executing 50 simple C statements.
4074 msgs/second with each message executing 10 simple C statements.

STATE EVENT HANDLING AS A MEASURE OF SCHEDULER SLOT DURATION

1000 msgs/second with each message executing 148 simple C statements, giving a task scheduling period of 1ms.
500 msgs/second with each message executing 323 simple C statements, giving a task scheduling period of 2ms.
250 msgs/second with each message executing 673 simple C statements, giving a task scheduling period of 4ms.
200 msgs/second with each message executing 848 simple C statements, giving a task scheduling period of 5ms.
100 msgs/second with each message executing 1,723 simple C statements, giving a task scheduling period of 10ms.
10 msgs/second with each message executing 17,473 simple C statements, giving a task scheduling period of 100ms.

Timings based on mean figures from motorola 68332 and intel 80386 systems @ 12MHz.

SYSTEM REQUIREMENTS

- Hardware reset counter for watchdog support.
- Primary programmable interval timer for timing utility support.
- Secondary programmable counter timer for scanner support (optional).
- Smallest configuration with message passing and task scheduling requires approximately 4K of memory.
- The system interval timer interrupt must have its vector address configured with an external reference to the procedure 'ISR_Timer'.
- ANSI C compiler, preferably with 'misra compliant' switch.
- VSMT-OS can be applied to most 8, 16 and 32-bit microprocessor target systems, including the following for example purposes only, but is by no means a definitive list;

ARM, ColdFire, PowerPC / Power Architecture, x86, and other processor families - 68000/1/8/10, 68302/306/307, 68330/6, 68340/9,360, 68020/30/40/60, 68HC11/16, 8086/186/188, 80286, 80386/486ex, V20/30/40/50/53, 80196, 80c51, C166, H8/500/300, 90C100/301, 80C166/165/167, 6502, 6809, Z80/180, NS32000 and many others.

DOCUMENT HIERARCHY AND PRODUCT COMPONENTS

VSMT-OS DESIGNERS HANDBOOK

An introduction to Virtual State Machine Technology and its approach to embedded system software development. It provides in addition, an overview of the features offered by VSMT-OS and the basic steps required when developing application software which is to function in a VSMT-OS environment. This is in fact, this document; **psvsmtos.doc**

USER INTERFACES - AIROS

This is a document providing an application programmers guide to all system services offered by VSMT-OS;   **uiairo.doc**

It covers the descriptions, usage and interfaces for all of the components offered by VSMT-OS; airoke, airort, airomm, airoqm, airogu and aiross.

USER INTERFACES - OTCTS

This is a document providing an application programmers guide to all test phase services offered by VSMT-OS;   **uiotct.doc**

It covers the descriptions, usage and interfaces for all of the test components offered by VSMT-OS; otctet, otctmp and otcttf.

LOGICAL COMPONENT SPECIFICATIONS

This is a set of documents providing complete detailed designs for each of the run-time and test-phase components. Intended to both support the process of understanding the source code or as construction manuals for hand-crafting your own implementation in the language of your choice.

For each of the AIRO components; **lcairoke**.doc, **lcairort**.doc, **lcairomm**.doc, **lcairoqm**.doc, **lcairogu**.doc, **lcaiross**.doc.

For each of the OTCT components; **lcotctet**.doc, **lcotctmp**.doc, **lcotcttf**.doc

VSMT-OS Software Source

This encompasses the entire implementation of VSMT-OS as a set of software source program files. VSMT-OS source files consist of the following;

Source File	Description
CPU specific	
sys_cpu.asm	Target cpu specific assembler procedures, generally not needed.
sys_cpu.h	Target cpu specific data type definitions and all code inserts macros.
VSMT-OS definition	
sys_def.h	System common data definitions.
sys_def.c	System configurable data areas.
vsmtos.h	OS specific data definitions
VSMT-OS startup	
vsmtmain.c	Start-up code linking target specific reset vector code with the OS and application software.
AIRO-KE	
airokeif.h	Kernel user interfaces.
airoke.h	Kernel exclusive data definitions.
airoke.c	Kernel source code.
AIRO-RT	
airortif.h	Real-time user interfaces.
airort.h	Real-time exclusive data definitions.
airort.c	Real-time source code.
AIRO-MM	
airommif.h	Memory management user interfaces.
airomm.h	Memory management data definitions.
airomm.c	Memory management source code.
AIRO-QM	
airoqmif.h	Queue management user interfaces.
airoqm.h	Queue management data definitions.
airoqm.c	Queue management source code.
AIRO-SC	
airoscif.h	Scanner user interfaces.
airosc.h	Scanner user data definitions.
airosc.c	Scanner source code.
AIRO-GU	
airoguif.h	General utilities user interfaces.
airogu.h	General utilities data definitions.
airogu.c	General utilities source code.
AIRO-SS	
airossif.h	System security user interfaces.
aiross.h	System security data definitions.
aiross.c	System security source code.

SOURCE FILE	DESCRIPTION
OTCT-ET	
otctetif.h	Event and sequence tracking user interfaces.
otctet.h	Event and sequence tracking data definitions.
otctet.c	Event and sequence tracking source code.
OTCT-MP	
otctmpif.h	System monitoring and performance user interfaces.
otctmp.h	System monitoring and performance data definitions.
otctmp.c	System monitoring and performance source code.
OTCT-TF	
otcttfif.h	Trace and fault creation user interfaces.
otcttf.h	Trace and fault creation data definitions.
otcttf.c	Trace and fault creation source code.

DOCUMENT HISTORY

Document Name:	Product specification / VSMT
Source Format:	MS-WORD
Source File:	ps-vsmt.doc

Original Author:	M C Willett
Date of Creation:	22nd March 1993
Date of Draft Review:	23rd June 1993
Date of Draft Release:	8th September 1993
Date of First Release:	17th December 1993
Date of Last Release:	4th November 2014