Teach your processor to fly with "VSMT-OS”, the smart path to real-time embedded software solutions.

Have you ever felt that the difficulties with software are not so much the complexities of the product features being developed but in trying to keep a grip on its critical mass. A problem is a problem and the software logic to solve a given problem is rarely unmanageable in itself. The dilemma begins when the various logical components becomes entangled and dependant on each other and the clean separation of functionality become vague. Scheduling of activities, prioritisation and timing of the demands of a real-time environment often complicate and burden the development effort and soon the new application being developed is inseparable from the systems software below it.

You’ve no doubt thought of using an embedded operating system but often or not, you make do with an in-house alternative. Familiar with the ‘noddy’ loop which grows and grows - then explodes, only to demand a re-design and thereafter never ceases to haunt you. Ever thought it would be nice to write only the software needed to respond to the real events in the product environment, and desired to partition designs into manageable and logical units of implemented code. And of promises of ease of testing and maintenance - and of effortless feature enhancement. The usual ‘solutions’ up until now have been probably too expensive, complicated and 3rd party reliant. Happily, there is a new alternative - all you need is VSMT-OS and a simplification in your design process.

VSMT-OS - Virtual State Machine Technology Operating Systems. A new way forward in embedded systems software development. A small, fast, and effective real-time systems core component. VSMT-OS allows you to maximise your processors capacity, minimise your applications software memory requirements and reduce the amount of new software needed to be developed. Simplify your designs and re-use components in subsequent products.

For perhaps the first time, you have an opportunity to design in a virtual process framework. Typically, you have a single processor on a PCB and many real-world processes to control with it. VSMT-OS allows you to design each process independently as if it was on its own exclusive processor environment. That is, on one processor board, you can have as many processes running concurrently as you like, and each process affording the ability to operate in as many virtual states as a process requires. For example, a process comprises 30 stages from start to finish, including successful and erroneous scenarios. Each stage is a sequence of actions appropriate to the stage in the process. Between each stage is a period of inactivity - waiting for a stage to complete or external events to occur. This period of inactivity is a 'state' in VSMT terminology. If a process is designed then implemented in software as a collection of states, bound together with process actions relevant for each stage, you have the basis of a virtual state machine (VSM). This VSM, is configured as a task and functions in it own virtual processor environment. VSMT-OS simply activates the appropriate ‘state’ corresponding to the current stage of the process sequence. Whilst active, the OS schedules that state as the current task in the domain of its VSM.

So, for every task assigned to a system process, you have unlimited numbers of virtual states which are themselves, virtual tasks. You have a design and implementation that not only mirrors the process taking place, you have the basis for a true event driven system. Code need only be written to carry out the actions necessary when specific events occur - code need only execute when the specific events occur. Specific Logic for scheduling, prioritising, queuing, and timing of these events along with logic to remember how and why a process stage was reached need no longer be developed.

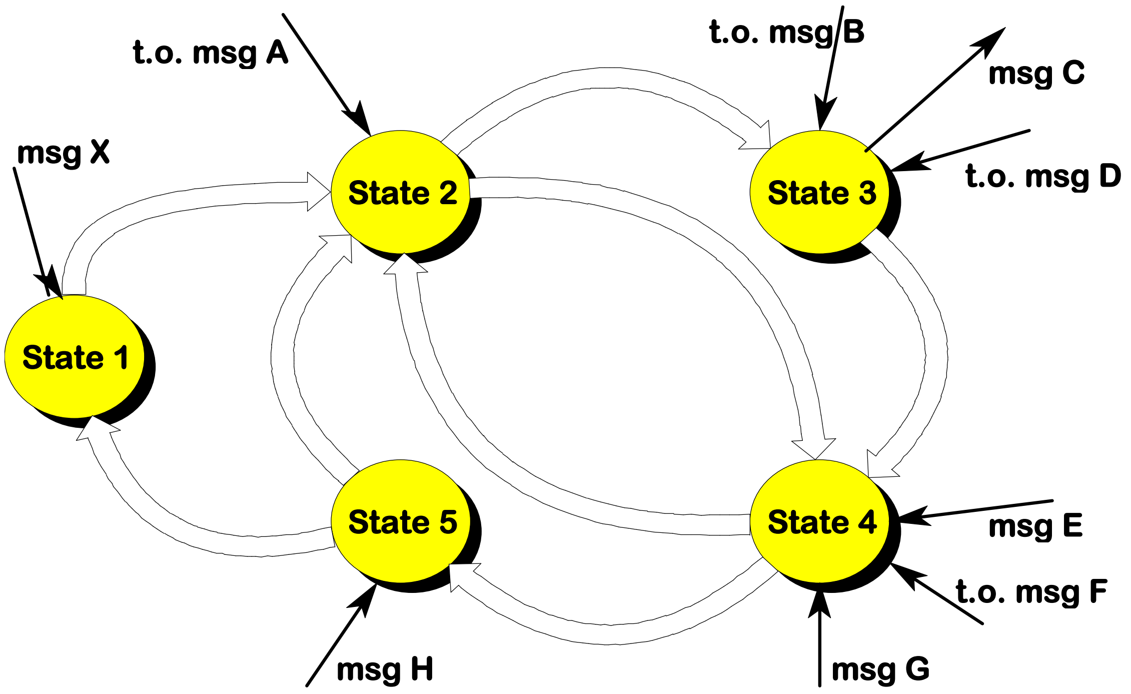
In any system all you are ever interested in is responding to user requests, responding to real-world hardware stimuli, and timing and controlling processes. VSMT-OS bases all communication of events occurring on the inter-task message. What VSMT-OS offers is total flexibility, total control and total performance. Since VSMT-OS forces scalable utilisation of system resources, nothing is wasted and the limiting factors for individual processor subsystems become the actual target processor’s MIPs and memory configured.

The scalability of a system is directly related to the individual requirements of a particular process. VSMT-OS forces each and every VSM to procure and own its system resources. If a new VSM won’t Link, it can’t run - when it does link, it will run! Therefore because memory requirements associated with a given resource are specified on an individual VSM basis at the design stage, you are able to determine exactly how much memory is required in advance. No more run-time ‘tweaking’ of system resources. So, what you get is the maximum number of processes possible on your target CPU, unlimited numbers of tasks comprising unlimited numbers of process states, each affording unlimited timing capacity.

In any state of any task for any process, you need only be interested in the messages appropriate to a state. All events, whether user requests, hardware events, or timing expiries, result in a message being sent to a task.

Any given message is therefore task to task, interrupt routine to task or timed event to task. Since timed events are processed by VSMT-OS, you need only request a timed event to subsequently receive notification of its occurrence. Timed events range from duration timers over periods of milliseconds to minutes and time related delivering prompts reminding tasks of time of day, elapsed time and regular events, whether they be single occurrence or continuous. Alarm systems, communications, television, music and consumer electronics are but a few product areas very much time and programme dependant. Today, there are very few products no longer concerned with the passage of time! Just imagine, a ticketing system. Five fare structures for seven time bands in every day. That is week-days, and not weekends, and not bank holidays..... and not Thursdays! Simple, request a timed event for each time band with the fare structure assigned to the event and wait! Let the other tasks sweat a little! Oh, and don't forget to write the procedures that deal with the new time band event messages received. And there it is, a chance to start developing new software rather than re-inventing wheels that aren't quite round.

A Virtual State Machine



Let's imagine the above diagram represents any single process; a communications protocol, a PBX telephone feature, or a washing machine wash programme to name but a few. We'll call it 'Process X'. Process X is one of many parallel processes in a given system. Process X becomes VSM X - Task X in VSMT-OS. Task X is a schedulable component in the VSMT-OS task environment. Process X comprises 5 stages which cover all possible scenarios through its process sequence. Task X will therefore have 5 states, one to handle each stage of the sequence. When a state is active, that state becomes a virtual task and the schedulable entity in the task domain of Task X. VSMT-OS will continue to schedule this particular virtual task according to Task X's scheduling allotment for however long the state is active. Of course, so long as jobs await Task X. Following through the diagram above, state 1 is configured as the first state to schedule for this Task X. In state 1, a unique message is expected before the process can commence. This may be as a result of user input or hardware event. In any case, message 'X' must be received and state 1 scheduled before a transition to state 2 can take place. Actions are then performed in state 2, and a time-out 'A' started. State 2 pauses, giving control back to VSMT-OS and is dormant until the next message received. Messages received always fall into 3 categories. Those that signify timed events, messages reporting state important user or hardware events, and messages which are deemed irrelevant in the context of the current state. In this case, state 2 starts a hardware process by driving i/o, times the process by starting a time-out for the appropriate period, then depending on the results of the process sequence, makes a transition to either state 3 or 4. Our scenario will follow a transition to state 3 where 2 process actions are required, the first for a few milliseconds, and the second for a few seconds. They must run in strict order so time-out 'B' is started and the state waits until scheduled again on time-out expiry when a message is supplied. A few actions are performed, one of which is the sending of a message 'C' to another task, and time-out 'D' then started. Task A goes to 'sleep' with the current state 3 being the virtual task now active in the OS scheduling environment. Timer 'D' expires, task A scheduled with state 3 receiving the time-out message, a few more actions performed and a transition to state 4 is made.

State 4 is now the schedulable entity and will not be scheduled until a hardware event occurs, it's interrupt routine driven and a message 'E' is sent. Once available, the OS schedules state 4 with message 'E' and time-out 'F' is started. If hardware event 'G' occurs before time-out 'F' expiry message occurs, the process sequence is continued according to plan, time-out 'F' can be cancelled, and a transition to state 5 can take place. If event 'G' does not occur first, i.e. time-out 'F' expiry message received before 'G', then a process sequence error has occurred and a transition back to state 2 must take place . If back in state 2, hardware event 'G' finally occurs, because it is no longer relevant, i.e. inappropriate to this new state, the message can simply be discarded. In state 5 following the successful path, the task waits doing nothing, occupying none of the processors bandwidth, until hardware event occurs and a notification message 'H' received. Depending on the message content, the process is either deemed successfully completed with a transition back to state 1 in readiness to start the process afresh or a transition to state 2 where recovery might be attempted.

VSMT-OS "the features"

Kernel Primitives
Get task identification • Reserve message buffer
Send message to • Wait for message
Receive message • Release message buffer

Real-time Primitives
Start time-out • Stop time-out • Request a wake-up
Request a regular prompt • Request reminder
Cancel request • Get system time and date
Set system time and date • Get absolute time
Get time-stamp • Check if summer time
Set local summer times • Check date not old
Check days in month • Validate time and date
Compare time dates • Add to time

Memory Management Primitives
Allocate buffer • De-allocate buffer

Queue Management Primitives (Linked Lists)
Initialise linked list • Append to top
Append to bottom • Append before last read
Read element from top • Read element from bottom
Read next element in list • Remove element from list
Remove element from top
Remove element from bottom

Queue Management (Cyclic Buffers)
Initialise cyclic buffer • Add to head • Add to tail
Read from head • Read from tail
Remove from head • Remove from tail

VSMT-OS "technical data"

Nr of tasks:	unlimited
RAM overhead:	60 bytes per task > 300 bytes system stack
Code overhead:	0.5KB - 15KB
Hardware needed:	CPU with 1x1ms timer interrupt
Nr of timers:	unlimited
Nr of concurrent task **timers**:	unlimited
Context switch time for a complete message passing transaction:	70 - 150 µseconds
Message queues:	1 per task, FIFOs with unlimited nr of entries
Memory management:	up to 6 pools, each with configurable buffer size and nr of buffers