MINNETRONIX™
MEDICAL

WHITE PAPER

Justification for the Queue-Based Task Model

Dennis Miller, Principal Software Engineer

Minnetronix has special technology expertise in embedded, safety-critical medical software and real-time operating systems. For new embedded software designs, we typically use a queue-based task model (QBTM) for best results. The benefits include reducing data contention, reducing coupling, increasing cohesion, and providing serialization. Each of these items improves the maintainability of the software and reduces software development time. An overview of the components of a QBTM, as well as an analysis of its advantages, drawbacks, and variations, illustrates why Minnetronix believes this model is often the best choice for exceeding each product's requirements.

What is a Task?

In order to fully understand the QBTM approach, a clear definition of tasks and queues is required. Tasks are independent sequences of instructions whose execution overlap over time. They can be thought to run simultaneously on a single core CPU. Tasks have a wide range of complexity. Four key types of tasks are described as follows:

1. A function that runs to completion each time it is called. The task function may be called by a cyclic executive; e.g. an infinite loop that repeatedly call a series of functions, or the task function may be called by a periodic timer interrupt handler.

2. A function that contains an infinite loop. This type of task is usually initiated by a scheduler. The scheduler switches between tasks, executing each in turn, according to a variety of strategies. The first two task types typically share the same memory space without memory protection. This allows them to easily share data, but they can unintentionally interfere with each other or develop complex interdependencies during development and maintenance.

3. A process. Processes are created by high level operating systems like Windows or Linux. Processes usually have independent memory spaces. Processes have advantages and disadvantages over tasks that share memory by default. A bug in one task can corrupt another task when those tasks share a single memory space. This may limit the effectiveness of the design goal of low coupling between modules. Processes are much less likely to corrupt other processes. This makes it easier to justify adequate segregation of modules with different IEC 62304 software safety classifications within a single processor. The corresponding disadvantage is that it takes more work to send data between processes.

4. A thread. A thread is a subtask within a process. Threads share the memory space of their parent process. Threads are similar in some ways with the first task type without memory protection.

What is a Queue?

In this context, a queue is a method for storing events for later action. The event that has been on the queue for the longest time is typically the next item removed from the queue. Queues are described as first-in, first-out (FIFO) data structures.

The critical feature of queues is providing a method of safely passing events and data between tasks. This is not a native feature of queues, but it is included in most implementations. Most Real-Time Operating Systems (RTOS) provide queues. Queues may also be implemented from scratch.

The number of items in a queue is limited in embedded systems to conserve memory and to avoid dynamic memory allocation. Each item usually has the same high-level type. However, the use of data structures and unions allows packing a wide variety of data within the high-level type. The size of each item is also limited to conserve memory.

What is the QBTM?

Each task defines and manages its own input queue. When possible, the input queue is private to the task. Each task defines access methods that build messages and place the messages on the task's queue. Other tasks do not need to know that the receiving task is using a private queue. Other tasks do need to know that responses may be delayed until the receiving task runs and handles the queued message.

Each task defines the internal message structure of the items on its queue. The data type usually includes a message identifier and a union of data structures.

- **Infinite Loop Tasks: Each task pends on its queue until a new message arrives. The task processes a single message from the queue and then returns to pending on its queue. When the task handles the message, it reads the message identifier to determine the type of event and which member of the union is valid.**

  This is the basic form of the model. Some tasks may use multiple queues for various purposes. A central feature of the model is that a task should pend on a single source, whether that source is a queue, semaphore, or some other single blocking function call.

- **Run to Completion Tasks: Run to completion tasks are similar to infinite loop tasks, but they check for new messages on their queue without blocking. These tasks do no work on a particular pass when there are no messages on their queue.**

What are the Benefits of the QBTM?

The benefits include reducing data contention, reducing coupling, increasing cohesion, and providing serialization.

- **Data contention is a classic computer science problem. The basic scenario is one task is in the middle of reading shared data when another task modifies the shared data. The first task ends up with a mix of old and new data when it finishes reading. Using queues as a primary method of data sharing provides the opportunity to focus on the correctness of the queue implementation, consequently preventing data contention for most data sharing. This approach also makes data sharing and event passing explicit. In contrast, obscure methods of data sharing are difficult to troubleshoot and maintain.**

- **Coupling is a measure of the interdependencies between software modules. Cohesion is a measure of how closely related items are within a software module. Coupling is usually contrasted with cohesion. This design reduces coupling by hiding the details of how each task implements its queue, or even whether the task actually uses a queue. It also increases cohesion by grouping functions related to sending data to a task within that task's source code modules.**

- **Serialization (or serializability) of events may simplify event processing within each task. Pure serialization may not be appropriate for some cases such as waiting for the completion of a previous action before handling some or all new events. Even in these cases, using this task model encourages making alternative event handling explicit, possibly by creating a finite state machine within the task.**

Each of these items should improve the maintainability of the software.

What are the Drawbacks of the QBTM?

Copying data into the queue is a good way to reduce data contention, but copying data has disadvantages. Queues need more memory to store data, which may be limited on an embedded system. It also takes CPU time to copy the data into the queue. One task putting data and events onto a queue and then a second task removing them from the queue is going to be slower than sharing unprotected global variables.

Since events sent to a task are queued, handling the events may be delayed, particularly for lower priority tasks. The main solution to this problem is designing the system so all tasks get the opportunity to run frequently.

Queues each support a limited number of items, so the queues may already be full when attempting to add a new item. Since the system should be designed so that queues are never expected to completely fill up, this is usually treated as a symptom of general software failure, leading to a controlled shutdown of the software.

Large amounts of data will not fit in the memory available for queues on most embedded systems. When large amounts of data must be shared alternative methods of sharing and sending events may be needed. A common method is to place a pointer to the data on the queue. However, this could lead to data contention problems since two tasks have access to the same memory. These problems are solvable. They just require additional software design and implementation.

Sometimes a calling task is requesting data from the receiving task, so a method for returning the data is needed. The parameters of the receiving task access function can include a callback function pointer. The receiving task defines the callback function signature. The calling task provides the callback function, which mostly likely builds a message and places the message on the original calling task's queue. However, this causes two sets of enqueuing and dequeuing which just increases the overhead of using queues. Also, the callback functions are trivial if the tasks share the same memory space. If the tasks do not share the same memory space, implementing callback functions becomes complex.

In some cases, the caller may need to have an immediate effect within the receiving task's module (for safety shutdown, exception handling, etc.). In another case, the caller may need a data value immediately for necessity or convenience (setting up a callback function and delaying the calling task when it just needs a simple Boolean value may be too inefficient). While sending requests through a queue to the task is the primary method, alternatives are possible. In any case, the task providing the access function is responsible for ensuring data contention is handled correctly.

ISRs may not be able to safely use the same functions for enqueuing messages or alerting a task that a new event has occurred. In some cases, alternative ISR-safe functions can be provided. An alternative way of alerting the task of new events may be necessary when no common method for the task to wait

for events from non-ISRs and ISRs exists. An alternative may be for the task to receive periodic timer tick event and to poll a global flag set by the ISR. Again, data contention around the global flag and related data needs to be addressed.

It is possible that the calling task may want to cancel a request that the receiving task has not acted on yet. However, the basic model does not support this.

The access functions, which write a message to a task's queue, actually run on the calling task's time and in the calling task's stack context. These functions should be kept short and simple.

Variations of the QBTM

Schedulers may be either preemptive or cooperative. Preemptive schedulers will switch from a lower priority task to a higher priority task when the higher priority task becomes ready to run. So it may be acceptable for a lower priority task to take a reasonably long time to complete an action, since other higher priority tasks will get to run.

Cooperative schedulers rely on each task yielding to allow other tasks to run. Cooperative tasks with lengthy operations to complete should split them into smaller operations, and yield between operations. The task will have to manage knowing it has remaining work to do, ensuring it gets scheduled to perform that work, and determine how to balance completing those operations with new requests.

Tasks can put messages on their own queue either for new or cascading actions, or to complete a lengthy operation. The task can use a single queue for its own internal event and external events, if those events are independent of each other. Otherwise, the task could have a separate queue for internal events. It would have to determine how to manage the two queues. One approach is to process all internal events before handling any new external events.

The queues do not have to be strictly FIFO. A task could look through its queue for particular events, and either give those events priority, or allow them to alter the handling of other events on the queue. For example, it might be appropriate to discard motor speed changes if a motor stop request is farther in the queue. However, when handling any events out of order, the task must ensure it will not act on requests in an inappropriate order. A task could also have a high priority queue and a low priority queue for different external independent events. Also, some queues support distinct priorities for each message on the queue.

Timer ticks for periodic processing or timeouts can be sent through the same message queue. Tasks can usually pend on a message queue indefinitely or with a timeout to detect error conditions, if periodic events are expected.

Software watchdog queries can be sent through the same message queue. In addition to showing the task is responsive; this also shows that the normal message queue processing is working.

Minnetronix and the QBTM

QBTM is a key component of Minnetronix' medical device software development approach. Minnetronix delivers products that span platforms and systems, all designed to exceed each product's requirements and meet the safety classification of each device. By utilizing industry best practices such as the QBTM, the Minnetronix development process strives to meet each customer's needs for performance, speed-to-market, and cost-effectiveness.

Minnetronix Medical, Inc.
www.minnetronixmedical.com