State-Event Executive

User Guide

Barnabas Projects Limited
60 Naishcombe Hill
Wick
Nr. Bristol BS30 5QS

Table of Contents

1. INTRODUCTION

1.1 Overview

(1) The State-Event Executive is a co-operative scheduling system which can be adapted to run on any suitable microprocessor. It is particularly suited to microcontrollers where RAM and particularly stack space is very limited, and a deterministic real-time response is required.

(2) The key feature that distinguishes this from other kernels is that all procedures must run to completion without calls to any 'wait' function to wait for an event or a time delay. This is the same as the normal practice used to design interrupt routines. It requires an object-orientated approach to design, in much the same way that modern Windows programs are designed, except that this system is much simpler, and the implementation is in C rather than C++.

(3) The advantage of this approach is that it is very efficient in its use of the stack and RAM. Task 'waiting' mid-way through in order to allow processing by a lower priority task, would require a significantly more complex context switch. On microcontrollers with very small stacks, this would entail significant manipulation of the stack, which may involve the shifting of stack segments in memory, making pointers to local variables become invalid without warning (a source of latent or unpredictable bugs).

(4) A further advantage is that all events are processed via a central point in the system, which makes for powerful tracing and debugging without the need to write special code.

(5) The system is table-driven which makes event response times very short.

(6) The executive revolves around event queues which exist at different priority levels. Processes post events which are then routed through the system and handled by the appropriate procedures which depend on the state machine to which that event belongs, and the state of that state machine.

(7) Thus, in a system where all the event functions are relatively small, and the longest is of a predictable length, then time-critical functions can be performed deterministically without the need to resort to pre-emptive scheduling.

(8) In a typical system, therefore, information is gathered in interrupt routines, and then processed at the appropriate priority by state-event processing routines. Note that it is possible to run a system completely without interrupts, though it usually preferable to have at least one timer interrupt routine for accurate scheduling of events.

(9) There are multiple versions of the executive compiled with different combinations of compile options. The options are:

(a) 8 or 16-bit events (define WORDQ to enable 16-bit events)
(b) Constant or variable structure (define VCONST= for variable structure)

(c) 8 or 16-bit control variables (define BIGAPP for 16-bit variables)

(d) Down counting timer (define COUNTDOWN for down counting)

(e) Data queues of greater than 255 bytes (define BIGDATA)

1.2 How to Use the Executive and How it Works

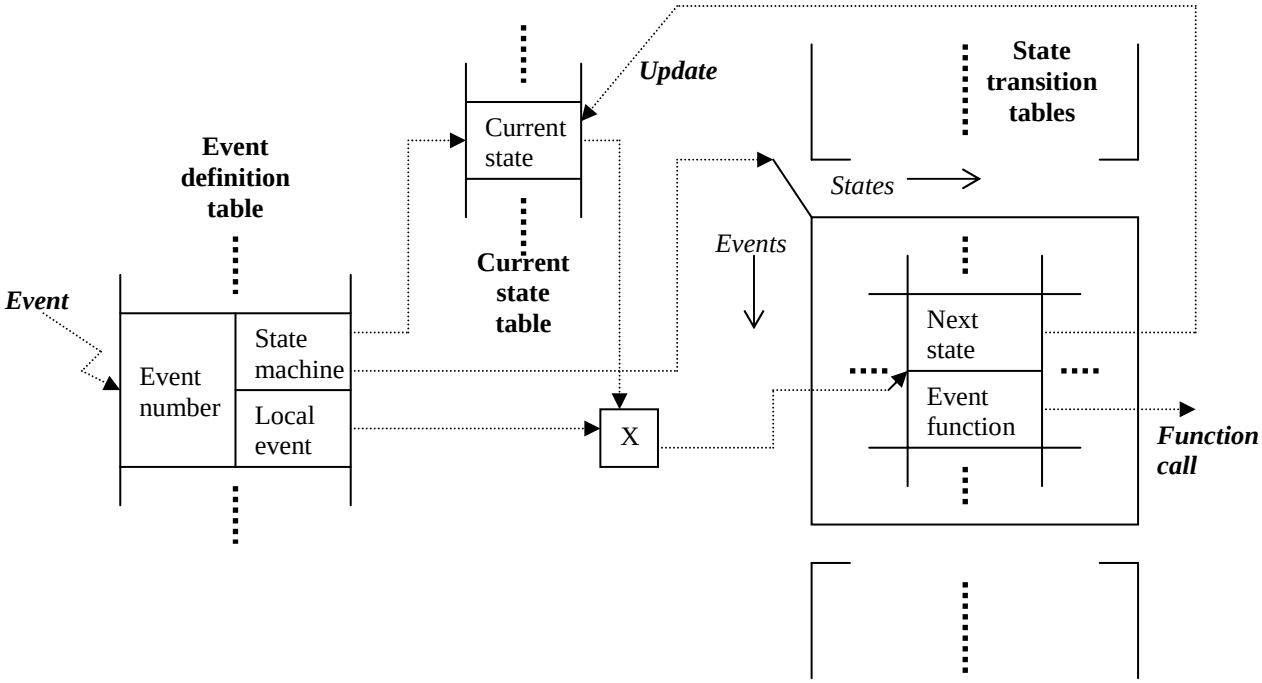
(1) The first and most important feature that must be appreciated is that applications written using the executive do not have a start and thread of execution in the conventional sense. There is no 'main' function (although this actually exists as part of the executive itself). Every item of processing that is performing is done in response to an event, with the exception of initialisation which is performed in an application-provided routine, called by the executive on start-up.

(2) Being table driven, an application does not merely consist of executable code, but also data tables of various kinds, most of which are designed to run in ROM. In addition to event-handling functions, each state machine has a state transition table, which defines the operation of a particular state machine, and how it hangs together.

(3) In addition to state transition tables which define the behaviour of state machines, there is a single main set of tables which define the events and state machines, and tie the system together. The system is also tied together by a set of definitions provided by the application header file (ExecApp.h).

(4) Other than event-handling functions, there are a number of other routines that have to be supplied in order for the application to successfully link with the executive. These functions handle initialisation, timers, system idle and debug trace data forwarding.

(5) The processing of an event and state transition can be illustrated as follows:



(6) For each state machine there are a fixed number of states defined. Events are all pre-defined by the application, and each event feeds one (and only one) state machine. For each combination of state and event on each state machine, a state to change to is defined, and also a function to be called as part of the state transition.

(7) No processing is performed without the posting of events, and a sequence of events is normally triggered by interrupt activity: the arrival of data of some sort. Each event is posted at a particular priority, and events are processed on a first-in-first-out basis and in order of priority.

(8) In terms of execution priority, there are a configurable number of priority levels (up to 256). Each priority level has an event queue associated with it. Thus, if there were 8 priority levels, there would be 8 event queues.

(9) If there is an event outstanding on a queue at a higher priority level, then the current event processing routine runs to completion, and after that processing is switched to the higher priority queue. All events are processed on that queue before returning to processing events on the original queue.

(10) The system is potentially capable of supporting up to 255 (optionally 65535) events and 256 (optionally 65536) state machines. Each state machine can be disabled or in one of up to 255 states, and be uniquely connected to up to 256 events each.

(11) All data structures that define a particular system structure (see section 2.5) are user-supplied. In terms of linkage, the names of the structures are referenced within the executive, and structure definitions exist within the executive header file, but the actual data is defined within the code of the application.

(12) A timer system is included as part of the executive. Timers are statically allocated and are single shot (not periodic). A portion of the implementation of the timer sub-system is provided by the application since it is hardware dependent. The system operates optimally when it uses a 16-bit free-running timer in conjunction with a 16-bit compare register, triggering an interrupt upon match. The application example (see section 4.5) shows a typical implementation. There may be up to 256 (optionally 65536) timers in a system.

(13) If an accurate periodic timer is required, then it is recommended to use a separate timer interrupt. Systems that use a sample cycle will often capture data in a periodic interrupt routine. The posting of an event from this interrupt routine is used to signal that data has been captured, and this is used as the basis of the main system sample period and scheduling.

(14) To ensure the integrity of these data structures, it recommended to use manifest constants (#defines) to define sizes of arrays, etc.

(15) Note that servicing of the watchdog is performed between the calling of event functions, and therefore should not be done in the application.

(16) For further clarification as to the structuring of an application, see the application example in section 4.

2. APPLICATION PROGRAM INTERFACE

2.1 System Calls

2.1.1 State Machine Functions

2.1.1.1 execCancelStateTransition()

(1) Function prototype:

```
void execCancelStateTransition(void);
```

(2) Parameters: none

(3) Returns: nothing

(4) For the currently executing state machine, the current state number is copied to the next state number.

(5) This function should not be called from an interrupt routine as it will cause unpredictable results.

2.1.1.2 execDisableStateMachine()

(1) Function prototype:

```
boolean execDisableStateMachine(word or byte Machine);
```

(2) Parameters: `Machine` state machine to be disabled

(3) Returns: true if it succeeded or false if it failed. It will fail if the parameter passed is invalid.

(4) This function sets this current state of a machine to zero (disabled).

(5) This function can be called at any time (including from within the state machine to be disabled).

(6) The current state table should not be accessed directly.

2.1.1.3 execEnableStateMachine()

(1) Function prototype:

```
boolean execEnableStateMachine(word or byte Machine,
                               byte InitialState);
```

(2) Parameters: `Machine` state machine to be enabled

`InitialState` initial state of that state machine

(3) Returns: true if it succeeded or false if it failed. It will fail if the parameters passed are invalid or if that state machine is already enabled, or if the state machine was disabled while it was being processed (which will always run to completion) and a request to enable was made while it was still being processed.

(4) The executive starts up with all the state machines disabled unless the current state table is defined by the application as having other initial values.

(5) If a controlled startup is required, then it is recommended to use this function.

2.1.1.4 execGetCurrentState()

(1) Function prototype:

```
byte execGetCurrentState(word or byte Machine);
```

(2) Parameters: `Machine` state machine to get current state of

(3) Returns: the state number of the specified state machine or 0 if an invalid state machine number is passed (0 also indicates that a state machine is disabled).

(4) This number is updated when the event processing function completes.

(5) The current state machine can be determined by calling `execGetStateMachine`.

2.1.1.5 execGetNextState()

(1) Function prototype:

```
byte execGetNextState(void);
```

(2) Parameters: none

(3) Returns: the state number that the currently being processed state machine will be set to when event processing function completes. If no state machine is being processed (this will only occur if this function is called from an interrupt routine or the idle routine) then zero will be returned.

2.1.1.6 execGetStateMachine()

(1) Function prototype:

```
word or byte execGetStateMachine(void);
```

(2) Parameters: none

(3) Returns: the number of the state machine which is currently being processed. If no state machine is being processed (this will only occur if this function is called from an interrupt routine, or if called from the application idle) then the number of the last state machine to have been processed will be returned.

2.1.1.7 execSetCurrentState()

(1) Function prototype:

```
boolean execSetCurrentState(word or byte Machine, byte NewState);
```

(2) Parameters: `Machine` state machine to set current state of
`NewState` the state to change the state machine to

(3) Returns: true if it succeeded or false if it failed. It will fail if the state specified is not a valid state for the state machine specified (or if the state machine number is invalid).

(4) Use this function for (temporary) debugging purposes only. Use `execCancelStateTransition` or `execSetNextState` to override the default state transition from within a state transition function. Use `execEnableStateMachine` to set the initial state of a state machine.

(5) Use of this function may give misleading results in event tracing.

2.1.1.8 execSetNextState()

(1) Function prototype:

```
boolean execSetNextState(byte NextState);
```

(2) Parameters: `NextState` the state to divert the transition to

(3) Returns: true if it succeeded or false if it failed. It will fail if the state specified is not a valid state for the current state machine.

(4) This function is used to override the 'next state' value in the state transition table, in order to divert the state transition to another state.

(5) This function should not be called from an interrupt routine as it will cause unpredictable results.

2.1.2 Event Processing Functions

2.1.2.1 execDeleteEvent()

(1) Function prototype:

```
boolean execDeleteEvent(word or byte Event);
```

(2) Parameters: `Event` posted event to be cancelled

(3) Returns: true if an event of that number was found and cancelled or false if not.

(4) This function searches all the queues from the highest priority (event at the front of the queue) to the lowest and substitutes an event number of zero in every occurrence it finds.

(5) It is recommended only to use this function if no other means of achieving this result can be done (i.e. it is best not to post the event in the first place if it is possible that it may need to be deleted). This is because it comparatively time consuming, although its efficiency is optimised by the fact that the event queues are contiguous, and therefore it just searches the entire array of all the queues from start to finish.

(6) Typically, this operation is performed when the source of an event is disabled (e.g. an interrupt routine or timer) and it is required to ensure that there are no events outstanding from that source.

2.1.2.2 execIsQueueFull()

(1) Function prototype:

```
boolean execIsQueueFull(byte Priority);
```

(2) Parameters: `Priority` indicates which queue

(3) Returns: true if the queue is full (cannot be posted to) or false if it is not. The same criteria are applied as per `execPostPriorityEvent`.

2.1.2.3 execPostEvent()

(1) Function prototype:

```
boolean execPostEvent(word or byte Event);
```

(2) Parameters: `Event` event number to post

(3) Returns: true if it succeeded or false if it failed. It will fail if the queue is already full or if the event is not valid.

(4) Events are processed in order of the priority of the queue on which it is placed.

(5) If the event queue system only supports byte events, then the high order byte of the event is ignored.

2.1.2.4 execPostInputEvents()

(1) Function prototype:

```
boolean execPostInputEvents(word NewInputs,
                            word *OldInputs,
                            word *ValidatedInputs,
                        struct execInputEvents *EventDefinitions);
```

(2) Parameters:

`NewInputs`	inputs at current poll (16 bits)
`OldInputs`	pointer to inputs at last poll
`ValidatedInputs`	pointer to filtered (validated) input states
`EventDefinitions`	pointer to input event definition table

(3) Returns: true if it succeeded or false if it failed. It will fail if any of the events in the event definition table are invalid (all valid events will be posted, however), or if any event could not be posted because its queue was full.

(4) This function is used to post events based on 16 bit inputs contained in a word.

(5) Only those bits indicated by 1's in the bit mask in the input event definition table are affected. This is so that bits within a particular input word can be processed at different scan rates and priorities.

(6) For each bit, the appropriate high or low event is generated if that bit in the new inputs is the same as in the old inputs but different from the validated inputs. This therefore implements single stage software debounce.

(7) If software debounce is not required, let `OldInputs` point to `NewInputs` as well, hence any change in state will appear to have occurred on successive iterations and will hence cause `ValidatedInputs` to change immediately on a change in state.

(8) When the events have been posted, `NewInputs` is copied to `OldInputs`.

(9) If a queue cannot be posted to, then the old inputs for the affected bits are not updated to the new values. That is, it will give further opportunities for the event to be posted when this function is next called with a new set of inputs (provided the relevant input does not change).

2.1.2.5 execPostPriorityEvent()

(1) Function prototype:

```
boolean execPostPriorityEvent(word or byte Event, byte Priority);
```

(2) Parameters:

`Event`	event number to post
`Priority`	queue priority to post to

(3) Returns: true if it succeeded or false if it failed. The same criteria are applied as per `execPostEvent`. Additionally, it will fail if an invalid queue priority is passed.

(4) The functionality is identical to `execPostEvent`, except that the event priority (and hence event queue) is overridden.

2.1.3 Timer Functions

2.1.3.1 execKillTimer()

(1) Function prototype:

```
boolean execKillTimer(word or byte Timer);
```

(2) Parameters: `Timer` the number of the timer to disable (kill)

(3) Returns: true if it succeeded or false if it failed. It will fail if an invalid timer number is passed (but not if the timer has already been killed).

(4) This disables the timer, but does not remove any pending events posted by that timer. Use `execPurgeTimer` if this is required.

2.1.3.2 execProcessTimeouts()

(1) Function prototype:

```
word execProcessTimeouts(word TickCount);
```

(2) Parameters: `TickCount` the current main timer value

(3) Returns: the tick count at which the next timeout is due to occur. If the value returned is the same as the value passed, it implies that all timers have timed out, and the timer subsystem can be disabled.

(4) This function is designed to be called from within the timer interrupt routine *only*. For this reason there is no queue locking performed on the timer queue.

(5) This function may be called without detrimental effects if no timeout is due. It may also be called late, that is, if a small number of tick counts have passed since the pending timeout has become due. For the sake of accuracy and performance, it is preferable, though, that this is called exactly at the point of timeout and only on the point of timeout.

2.1.3.3 execPurgeTimer()

(1) Function prototype:

```
boolean execPurgeTimer(word or byte Timer);
```

(2) Parameters: `Timer` the number of the timer to disable (kill)

(3) Returns: true if it succeeded or false if it failed. It will fail if an invalid timer number is passed (but not if the timer has already been killed).

(4) This disables the timer and removes any pending events posted by that timer. (Note that all occurrences of the event stored as the EventToPost for that timer will be removed from the event queues. The implication is that this event would be unique to this timer, i.e. not used by any other timer or function. Use `execKillTimer` in preference to this function unless this action is definitely required. This is because removing events from the queues is time consuming.

2.1.3.4 execSetTimer()

(1) Function prototype:

```
boolean execSetTimer(word or byte Timer, word TimeDelay,
                                          word or byte Event);
```

(2) Parameters:

`Timer`	the number of the timer to set
`TimeDelay`	the value of the time delay
`Event`	the event number to post on timeout

(3) Returns: true if it succeeded or false if it failed. It will fail if an invalid timer or event number is passed.

(4) Setting a time delay or event number of 0 will disable the timer. The macro `execKillTimer` is provided for this purpose.

(5) Calling an already-running timer will restart it with the new time delay value and event.

(6) Note that the timer units are user defined, and depend on the programming of `execAppTimerControl` and its associated interrupt routine.

2.1.4 Data Queue Functions

2.1.4.1 execDataQueueOk()

(1) Function prototype:

```
boolean execDataQueueOk (byte *Queue, word or byte QueueSize);
```

(2) Parameters:

`Queue`	the queue to get the data from
`QueueSize`	the size of the queue in bytes

(3) Returns: true if the queue is not corrupted. It will return false if the queue's internal data structures (pointers, etc.) imply that queue entries exist outside the memory extent of the queue or overlap one another.

(4) The `QueueSize` is the size of the `Queue` array in bytes, which is best obtained using the `sizeof()` operator. This may be most easily done using the `DATAQ()` macro, where `DATAQ(MyQueue)` expands to `MyQueue, sizeof(MyQueue)`. Note that `MyQueue` must be defined by `extern byte MyQueue[`*x*`]` not `extern byte *MyQueue` in the application's header file.

(5) For details of the queue implementation, see section 3.4.4.

2.1.4.2 execGetData()

(1) Function prototype:

```
byte execGetData (byte *Queue, word or byte QueueSize,
                                                byte *Buffer);
```

(2) Parameters:

`Queue`	the queue to get the data from
`QueueSize`	the size of the queue in bytes
`Buffer`	the buffer in which to place the retrieved data

(3) Returns: the number of bytes retrieved. It will return 0 if there is nothing pending on the data queue.

(4) The buffer must be big enough to hold the data retrieved (no check is made). The number of bytes retrieved will never exceed 255.

(5) The `QueueSize` is the size of the `Queue` array in bytes, which is best obtained using the `sizeof()` operator. This may be most easily done using the `DATAQ()` macro, where `DATAQ(MyQueue)` expands to `MyQueue, sizeof(MyQueue)`. Note that `MyQueue` must be defined by `extern byte MyQueue[`*x*`]` not `extern byte *MyQueue` in the application's header file.

(6) For details of the queue implementation, see section 3.4.4.

2.1.4.3 execGetDataItems()

(1) Function prototype:

```
byte or word execGetDataItems (byte *Queue,
                                    word or byte QueueSize);
```

(2) Parameters:

`Queue`	the queue in which to count items
`QueueSize`	the size of the queue in bytes

(3) Returns: the number of items (individually posted buffers) that are present on the queue. It will return 0 if the queue is empty.

(4) The `QueueSize` is the size of the `Queue` array in bytes, which is best obtained using the `sizeof()` operator. This may be most easily done using the `DATAQ()`

macro, where `DATAQ(MyQueue)` expands to `MyQueue, sizeof(MyQueue)`. Note that `MyQueue` must be defined by `extern byte MyQueue[`*x*`]` not `extern byte *MyQueue` in the application's header file.

(5) For details of the queue implementation, see section 3.4.4.

2.1.4.4 execGetDataSpace()

(1) Function prototype:

```
byte execGetDataSpace (byte *Queue, word or byte QueueSize);
```

(2) Parameters:

`Queue`	the queue to examine for space
`QueueSize`	the size of the queue in bytes

(3) Returns: the number of bytes that can be posted onto the given data queue. It will return 0 if the data queue is completely full.

(4) The `QueueSize` is the size of the `Queue` array in bytes, which is best obtained using the `sizeof()` operator. This may be most easily done using the `DATAQ()` macro, where `DATAQ(MyQueue)` expands to `MyQueue, sizeof(MyQueue)`. Note that `MyQueue` must be defined by `extern byte MyQueue[`*x*`]` not `extern byte *MyQueue` in the application's header file.

(5) For details of the queue implementation, see section 3.4.4.

2.1.4.5 execPeekData()

(1) Function prototype:

```
byte execPeekData (byte *Queue, word or byte QueueSize,
                   byte *Buffer, word or byte ItemNumber);
```

(2) Parameters:

`Queue`	the queue to get the data from
`QueueSize`	the size of the queue in bytes
`Buffer`	the buffer in which to place the retrieved data
`ItemNumber`	the buffer from the front of the queue to inspect

(3) Returns: the number of bytes retrieved. It will return 0 if the item number does not exist (i.e. is outside the bounds of the number of items on the queue).

(4) The buffer must be big enough to hold the data retrieved (no check is made). The number of bytes retrieved will never exceed 255.

(5) When the data is retrieved, only a copy of the data is taken. The queue itself remains unaltered.

(6) Item number 0 refers to the item that will be fetched in the next `execGetData()` call. Item numbers then follow sequentially along the queue from there, up to the most recently posted.

(7) The `QueueSize` is the size of the `Queue` array in bytes, which is best obtained using the `sizeof()` operator. This may be most easily done using the `DATAQ()` macro, where `DATAQ(MyQueue)` expands to `MyQueue, sizeof(MyQueue)`. Note that `MyQueue` must be defined by `extern byte MyQueue[`*x*`]` not `extern byte *MyQueue` in the application's header file.

(8) For details of the queue implementation, see section 3.4.4.

2.1.4.6 execPendingData()

(1) Function prototype:

```
byte execPendingData (byte *Queue);
```

(2) Parameters:

`Queue`	the queue on which to check whether there is data

(3) Returns: the number of bytes pending retrieval. It will return 0 if there is nothing pending on the data queue.

(4) For details of the queue implementation, see section 3.4.4.

2.1.4.7 execPostData()

(1) Function prototype:

```
boolean execPostData (byte *Queue, word or byte QueueSize,
                                   byte *Buffer, byte Size);
```

(2) Parameters:

`Queue`	the queue on which to post the data
`QueueSize`	the size of the queue in bytes
`Buffer`	the data to post
`Size`	the number of bytes of data to post

(3) Returns: true if it succeeded or false if it failed. It will fail if there is not enough space on the queue to post the data.

(4) The `QueueSize` is the size of the `Queue` array in bytes, which is best obtained using the `sizeof()` operator. This may be most easily done using the `DATAQ()` macro, where `DATAQ(MyQueue)` expands to `MyQueue, sizeof(MyQueue)`. Note that `MyQueue` must be defined by `extern byte MyQueue[`*x*`]` not `extern byte *MyQueue` in the application's header file.

(5) For details of the queue implementation, see section 3.4.4.

2.1.4.8 execPurgeData()

(1) Function prototype:

```
void execPurgeData (byte *Queue);
```

(2) Parameters: `Queue` the queue to be purged

(3) Returns: nothing.

(4) This function resets a queue and deletes all its contents. (In actual fact, only the header data is reset. The remaining bytes are untouched.)

2.1.5 Asynchronous Communications Protocol Functions

2.1.5.1 Important Points

(1) The protocol is designed for use on multi-drop master-slave arrangements such as RS485. The protocol supports a master and up to 254 slave nodes (although RS485 physically supports only up to 32 nodes on a given link). The protocol also supports broadcast, which is implemented by transmitting on a node number of 255. The protocol can also be used with a point-to-point arrangement such as RS232.

(2) The protocol uses a half-duplex poll-response system, where the master polls one node at a time, and that node responds before the master polls that node again or moves onto another one. The only exception to this is broadcast, where the master sends out a message without expecting a response. The format of the message in each direction is as follows:

DLE	STX	Node	SeqNo	Data0 … DataN	DLE	ETX	16-bit Checksum

(3) The protocol is fault-tolerant in that it supports message retries, and the sequence number (SeqNo above) is provided for this purpose. The master controls the sequence number, and it is recommended that when the master starts, its first message uses a sequence number of 0. Subsequent messages then increment the sequence number wrapping round after 255 to 1 (not 0). Slaves respond with the same sequence number as the message to which they are responding. If a sequence number the same as the previous, then it was because the master did not receive the reply correctly (it is retrying), and the last reply should be re-transmitted without processing the incoming data. A sequence number of 0 indicates a start-up and slaves should therefore always process the message and give a fresh response for this sequence number.

(4) In order for the DLE-STX and DLE-ETX sequences to be unique, any data byte in the data portion of the message that is a DLE byte is followed by a further DLE byte. The node and sequence number do not conform to this rule.

(5) The checksum is a CRC-16 of all the bytes between (and not including) the DLE-STX and the DLE-ETX.

(6) If a message arrives while one is being transmitted (and it is not listening to its own outgoing message), then there is a sequencing problem. On the slave, it is advisable not to respond in this situation, and on the master it is advisable to wait for another response to come in before proceeding.

(7) In systems where the same transmission medium is used for transmission and reception (such as two-wire RS485 or single-frequency radio) there may be a problem with turnaround time, in that slaves may respond too quickly, before the master has had a chance to switch hardware modes from transmission to reception. This is particularly likely where the master is a non-embedded computer such as a PC running Windows or Unix. In this situation, the slave must delay its response, or separate media for transmit and received must be used, such as 4-wire RS485, where the master transmits on one pair and receives on another.

(8) When communicating between dissimilar systems, issues of byte ordering and structure packing become relevant. Routines are provided to normalise the data format over the transmission medium, to solve this particular problem. A fully packed (1-byte aligned) big-ending format is used (big-endian = high order byte first). This is the easiest format to interpret on a communications analyser, and is commonly used on many different types of network.

2.1.5.2 execAsyncBufferOut()

(1) Function prototype:

```
word execAsyncBufferOut (byte Node, byte *Raw, byte *TxBuf,
                                    byte RawSize, byte SeqNo);
```

(2) Parameters:

`Node`	the target (recipient) node number
`Raw`	the unformatted data to be transmitted
`TxBuf`	the buffer to receive the formatted data
`RawSize`	the size of the unformatted data
`SeqNo`	the sequence number of the message

(3) Returns: the number of formatted bytes placed in `TxBuf`.

(4) This function puts the data to be transmitted into a protocol packet as per the protocol described in section 2.1.5.1.

(5) The maximum size of the formatted data is `2*RawSize + 8`. This assumes that all the data bytes are 0x10 (DLE). Clearly, this is unlikely to be the case in practice, and a sufficiently large buffer should be provided to cover all eventualities of the application in question.

2.1.5.3 execAsyncByteIn()

(1) Function prototype:

```
boolean execAsyncByteIn (byte Value,
                  struct execAsyncStateDef *State, byte *Buffer);
```

(2) Parameters:

`Value`	the byte just received, to be processed
`State`	the state and progress of the incoming data stream
`Buffer`	the buffer receiving the incoming data

(3) Returns: true when a complete or corrupted message has been received, false otherwise.

(4) This routine is designed to be called from within the receive interrupt routine, where bytes are received one at a time.

(5) See section 2.5.6 for a description of `State` and how the information should be used.

(6) This function places only processed bytes in the buffer provided. All the protocol has been stripped by the time data has finished arriving in this buffer.

2.1.5.4 execPackData()

(1) Function prototype:

```
word execPackDataEx (byte *Target, byte *Source,
                        struct execStructDef VCONST *Descriptor);

word execPackData (byte *Buffer,
                        struct execStructDef VCONST *Descriptor);
```

(2) Parameters:

`Target`	the buffer where the packed data is to be saved
`Source`	the buffer containing the data to be packed
`Buffer`	the buffer for packing the data in-place
`Descriptor`	an array of structure item description structures

(3) Returns: the size of the packed data in bytes.

(4) This routine reformats the data provided into a format for transmitting over a network or a data link. All structure padding is removed and the bytes are re-ordered (if necessary) to be big-endian. In the second version (above) of this function, the transformation is done in-place, and the old data is therefore effectively overwritten.

(5) The structure and layout of the descriptor array is described in section 2.5.6.

2.1.5.5 execUnPackData()

(1) Function prototype:

```
word execUnPackDataEx (byte *Target, byte *Source,
                  struct execStructDef VCONST *Descriptor);

word execUnPackData (byte *Buffer,
                  struct execStructDef VCONST *Descriptor);
```

(2) Parameters:

`Target`	the buffer where the unpacked data is to be saved
`Source`	the buffer containing the data to be unpacked
`Buffer`	the buffer for unpacking the data in-place
`Descriptor`	an array of structure item description structures

(3) Returns: the size of the unpacked data in bytes.

(4) This routine reformats the data provided from a format for transmitting over a network or a data link into a format that corresponds to the structure local to the CPU/compiler. The relevant structure padding is and the bytes are re-ordered (if appropriate to the CPU) to be little-endian. In the second version (above) of this function, the transformation is done in-place, and the old data is therefore effectively overwritten.

(5) The structure and layout of the descriptor array is described in section 2.5.6.

2.1.6 System and Other Functions

2.1.6.1 execCheckIntegrity()

(1) Function prototype:

```
byte execCheckIntegrity(byte WhatToCheck);
```

(2) Parameters: `WhatToCheck` checks as defined below

(3) Returns: sum of failed checks as below (0 = passed).

(4) The following checks can be performed:

(a) Do all the (global) events belong to a valid state machine? (1)

(b) Do all the (global) events correspond to a valid local event within its state machine? (2)

(c) Do all the events have valid default priorities? (4)

(d) Do all the state transitions in the state machines call up valid states for that state machine (0 is invalid)? (8)

(e) Are all the queues contiguous? (16)

(f) Are all the queue counts and tails within the bounds of the queue sizes? (32)

(g) Are the current states of the state machines within the bounds of the number of states in each state machine (0 is valid, state machine disabled)? (64)

(h) Are all the active timers part of the active timer queue? (128)

(5) When specifying what to check, pass 0 for all the checks to be performed, or add together the figures in brackets after the questions above for the tests that are required to be performed.

(6) For each test, if it fails, the number in brackets after the questions above are added to the return value to give a failure category bitmap.

(7) Normally this function is called during the initialisation function to verify that the application and its data tables have been built correctly, although the last two checks verify the state of data in RAM and can be called at any time.

2.1.6.2 execGetPriority()

(1) Function prototype:

```
byte execGetPriority(void);
```

(2) Parameters: none

(3) Returns: the current priority level. If the system is idling, then 0 is returned. Note that 0 is also the lowest priority.

2.1.6.3 execSafeCpuIdle()

(1) Function prototype:

```
void execSafeCpuIdle(void);
```

(2) Parameters: none

(3) Returns: nothing

(4) This function is designed to be called from within `execAppIdle`. It puts the CPU in idle mode, checking first that there is no processing outstanding. It does this in a single uninterruptable instruction sequence, so that it is impossible for an interrupt routine to trigger off processing between deciding that idling is appropriate and actually going into idle. If this were to happen, the system may stall with pending events, possibly ending in a watchdog reset.

(5) One means of monitoring the level of CPU usage is to set a port pin just prior to calling this and to reset it afterwards. Monitoring this pin on an oscilloscope will then show the periods of time that the CPU is in idle.

2.1.6.4 execTraceEvent()

(1) Function prototype:

```
boolean execTraceEvent(boolean Enable, word or byte Event);
```

(2) Parameters:

`Enable`	true to enable, false to disable
`Event`	event number to trace (0 = global)

(3) Returns: true if it succeeded or false if it failed. It will fail if the event is not a valid event number.

(4) This function is used to enable/disable event tracing on a particular event or all events.

(5) Event 0 is used to globally enable/disable event tracing.

2.1.6.5 execUpdateChecksum()

(1) Function prototype:

```
word execUpdateChecksum(byte Value, word Checksum);
```

(2) Parameters:

`Value`	byte to update checksum with
`Checksum`	checksum to be updated

(3) Returns: the updated checksum.

(4) This function implements a CRC-16 algorithm using a lookup table. It is used principally by the asynchronous communications functions, but is available for general usage.

(5) To calculate a CRC-16 checksum, call this routine repeatedly, iterating over the buffer to be checksummed. Start the process with a checksum value of zero.

2.2 Application-Provided Routines

2.2.1.1 execAppCurrentTick()

(1) Function prototype:

```
word execAppCurrentTick(void);
```

(2) Parameters: None

(3) Returns: the current value of the main timer (the current timer tick).

(4) This function is *not* part of the executive but is user-supplied and called by the executive when a timer is being activated.

(5) This function should query the hardware or other timer/counting mechanism used to implement the timer system.

2.2.1.2 execAppFinish()

(1) Function prototype:

```
void execAppFinish(void);
```

(2) Parameters: None

(3) Returns: Nothing

(4) This function is *not* part of the executive but is user-supplied and called only when running under Windows when the executive shuts down.

(5) It is not necessary to provide this function when running only in an embedded environment.

(6) Use this function to de-initialise and release any resources allocated in `execAppInit()`.

2.2.1.3 execAppIdle()

(1) Function prototype: `void execAppIdle(void);`

(2) This function is *not* part of the executive but is user-supplied and called by the executive repeatedly while there are no events to process.

(3) This function can be used to set the processor into the idle state, or to toggle an output pin or do other calculations to measure how much spare processing bandwidth is available under different operating conditions.

2.2.1.4 execAppInit()

(1) Function prototype: `void execAppInit(void);`

(2) This function is *not* part of the executive but is user-supplied and called by the executive before any other processing is performed.

(3) In this function, initialise hardware, state machines and perform all other initialisation, and enable the interrupt system.

(4) The executive includes a `main()` function, so do not create one in the application program.

2.2.1.5 execAppPostExternalEvent()

(1) Function prototype:
```
boolean execAppPostExternalEvent(word Event,
                                 byte Channel);
```

(2) Parameters:

`Event`	the event number to post
`Channel`	the channel to post the event to

(3) Returns: true if the event was successfully posted.

(4) This function is *not* part of the executive but is user-supplied and called when an event is posted by the application with a priority outside the range of the event queues.

(5) The channel number is zero-based and is the 'priority' of the posted event minus the number of event queues in the system.

(6) This function should pass on or use the event if it can, returning true, or returning false if any of the parameters are out of range or if the action cannot be performed for any reason.

2.2.1.6 execAppResyncTimer()

(1) Function prototype:

```
void execAppResyncTimer(word TickCount, boolean Enable);
```

(2) Parameters:

`TickCount`	the time at which the next timeout will occur
`Enable`	true if there are timeouts pending

(3) Returns: nothing.

(4) This function is *not* part of the executive, but is user supplied.

(5) It is called when a timer is being set up or disabled, and is only called if the time at which the next (pending) timeout was due to occur, has changed.

(6) If `Enable` is false, then there are no more pending timeouts, and hardware timers may be shut down. No further timer interrupts are required (until this function is called again with `Enable` true). Additionally, in this case, the value of `TickCount` is arbitrary, and should not be used.

(7) If `Enable` is true, the hardware timer system should be reset or modified so that the next timeout (timer interrupt) occurs at the new `TickCount` value, rather than the one that was currently due.

(8) In this way, this routine should co-operate with a user-supplied timer interrupt routine which should determine when the selected timer has reached its timeout value. At the point it determines this, it should call `execProcessTimeouts()` and set up the next time to time out or disable itself (depending on the return value from this function – see section 2.1.3.2).

2.2.1.7 execAppTrace()

(1) Function prototype:

```
void execAppTrace(byte BeforeState, byte AfterState,
                                          word or byte Event);
```

(2) Parameters:

`BeforeState`	state before the state transition
`AfterState`	state after the state transition
`Event`	event number causing the transition

(3) Returns: nothing.

(4) This function is *not* part of the executive, but is user supplied.

(5) It is called when event tracing is enabled (globally) and that particular event is enabled for tracing. It is called after the state transition function just before the current state is set to the next state.

(6) Typically this function will store the information passed in a circular buffer and/or send it down a communications link.

(7) Care should be taken when tracing frequent events so as to avoid overloading the system and affecting performance.

2.2.1.8 Timer Interrupt Routine

(1) See the description for `execAppResyncTimer` in section 2.2.1.6.

2.3 Running under Microsoft Windows

2.3.1 Important Points

(1) The entire executive will compile and run under Windows without modification, and additionally requires the module ‘ExecWin.c’ which defines Windows-specific functionality.

(2) The Windows-specific file has been designed for compilation under Microsoft Visual C/C++, although it may work with other Windows C compilers.

(3) The executive and application must be compiled and linked using the multithreaded C libraries (a non-default option in Microsoft C/C++).

(4) The `execAppIdle()` function must call the `execSafeCpuIdle()` function, because shutdown of the executive is performed from this function (and it also allows the proper yielding of execution in the Windows pre-emptive multitasking environment). Without this, the executive will not function properly under Windows.

(5) The `execAppCurrentTick()`, `execAppResyncTimer()` functions and the equivalent of the timer interrupt routine is implemented within the Windows module and should not be defined in the application (use `#ifndef _WIN32` in the embedded application code to prevent any embedded application-provided versions of these from being compiled under Windows). The size of the timer tick (in integral multiples of 100 nanoseconds) is assigned when starting up the executive under Windows, and thus an application designed for embedded use may run in a simulated environment under Windows in real time. (Note that the timer system under Windows 95, and subsequent DOS-based versions of Windows, may be using with 18ms DOS timer interrupt. Although cumulative errors may be minimal, precise short time delays may not be achievable).

(6) Use the `execWinStart()` and `execWinQuit()` functions to start and stop the executive. The executive may be started and stopped on multiple occasions during the run time of a program, and each time the executive is started, it completely re-initialises itself.

(7) Once started, the executive runs in a background thread. This means that the foreground part of the program, which started it, behaves towards it as if it were an interrupt routine. Therefore, from the foreground thread (typically consisting of Windows GUI event handling functions), only use functions which are safe to be called from within interrupt routines, such as `execPostEvent()`, unless the entire executive is explicitly locked.

(8) The executive may be locked and unlocked using `execWinLockExec()` and `execWinUnLockExec()`. Locking the executive waits until the executive is idling before locking out the executive's thread. The executive's thread is then blocked until the executive is unlocked.

(9) Only one instance of the executive may run at any one time, and calling `execWinStart()` multiple times without corresponding calls to `execWinQuit()` will have no effect.

(10) If `VCONST` (variable constants) is defined in the compile options, then the executive may be re-configured at run time. Clearly, it is inadvisable to alter the structure of the application, which runs under the executive, while the executive is running.

(11) When running under Windows, the executive additionally requires the application to provide the function `execAppFinish()`. This is called as the last item when the executive shuts down, after all outstanding events have been processed.

2.3.2 execWinLockExec()

(1) Function prototype:

(12)
```
void execWinLockExec ();
```

(2) Parameters: none

(3) Returns: nothing

(4) This function is used to lock the executive thread so that another thread can safely access any variables belonging to an application running under the executive.

(5) This function waits until the executive is idling before locking the thread and returning. If the executive is continually processing events, then this function will never return.

(6) For each call of this function, there must be a corresponding call to `execWinUnLockExec()`, otherwise the executive will be permanently locked out and will also never exit safely.

2.3.3 execWinStart()

(1) Function prototype:

```
boolean execWinStart (lword TickSize,
                      lword StackSize,
                      slword Priority);
```

(2) Parameters:

`TickSize`	the size of the timer tick in 100ns intervals
`StackSize`	the size of the stack used by the executive's thread
`Priority`	the priority of the executive's thread

(3) Returns: true if it succeeded or false if it failed. It will fail an instance of the executive is already running or if any of the Windows resources used by the executive fail to be created.

(4) This function starts the executive in a background thread in the Windows environment.

(5) Setting a tick size of zero will allocate a 1 millisecond timer tick.

(6) Setting a stack size of zero will allocate a default stack size, which will be the same as the main foreground application's stack size.

(7) Setting a priority of zero will select 'normal' priority which is the default thread priority under the process priority of the current application. Use the appropriate Windows manifest constants for thread priority assignment.

2.3.4 execWinQuit()

(1) Function prototype:

```
boolean execWinQuit (lword Timeout);
```

(2) Parameters: `Timeout` time in milliseconds to wait for termination

(3) Returns: true if it succeeded or false if it failed. It will fail if the executive is not running.

(4) This function causes the executive to shut down in an orderly fashion. Firstly, the timer thread shuts down so that no more timer events are posted, and then all events are processed, before the executive itself finally exits.

(5) If the timeout time is reached before the executive has fully shutdown, its thread is immediately terminated, and various items in the system associated with the thread may be left in an indeterminate state. Therefore, timing out should be considered as a very last resort, and the timeout should be very long or `INFINITE` (manifest constant).

2.3.5 execWinUnLockExec()

(1) Function prototype:

```
void execWinUnLockExec();
```

(2) Parameters: none

(3) This function unlocks the executive thread and allows its event processing to continue.

(4) It should be called after a corresponding call to `execWinLockExec()`.

2.4 Test Harness Facilities

2.4.1 Introduction

(1) Applications written using the State-Event Executive can be tested in a scripting or other controlled test environment with the aid of the facilities described in this section.

(2) A special version of various application-defined facilities is used to enable this, as defined in `ExecTest.c` and `TestDep.h`. This allows timers and events to be progressed under full control test facility itself, so that testing is not subject to the passage of real time, and so that the state of a system's internal data can be verified at any point in its sequence of events.

(3) Hence, the test facility implements execAppCurrentTick(), execAppResyncTimer() and the safe CPU idle code. It also provides a means of breaking in and out of the executive's main loop and re-starting and application after it has been running.

2.4.2 execTestAdvanceTimer()

(1) Function prototype:

```
void execTestAdvanceTimer (word TickCount);
```

(2) Parameters: `TickCount` time (in ticks) to manually advance time by

(3) Returns: nothing.

(4) This function advances the system timer by the specified increment, causing the posting of events (in sequence) of any timers that time out during that time increment.

(5) Note that the timer in the test environment is not tied to any real timer, and the only way to progress simulated time is via this function.

2.4.3 execTestDoEvents()

(1) Function prototype:

```
word execWinQuit (word Events);
```

(2) Parameters: `Events` the maximum number of events to process

(3) Returns: the number of events actually processed.

(4) Call this function to run the main loop of the executive as many times as is required to either process all the events in the event queues, or to process the number of events specified, if this number is less than the number of events pending.

(5) Calling this function does *not* perform any initialisation (`execAppInit()`). `execTestStart()` should be called prior to any number of calls to this function.

(6) Call this function with a value of `0xffff` to process all outstanding events.

2.4.4 execTestStart()

(1) Function prototype:

```
void execTestStart (void);
```


(2) Parameters: none.

(3) Returns: nothing.

(4) This function initialises the executive's own internal data structures and calls the application initialisation function (`execAppInit()`).

(5) It cannot initialise any of the application's global (or other static) data. Therefore, if this function is being called subsequent to exercising part of the system, then the test environment must also re-initialise the application's data.

2.5 Application-Provided Data Structures

2.5.1 Event Definition Table

(1) This is defined as follows:

```
struct execEventDef
{
      byte DefaultPriority;
      byte LocalEventNumber;
      word or byte StateMachineNumber;
};

struct execEventDef execEvent[TOTAL_EVENTS];
```

(2) Each event is given a unique number in the system which is either a byte or a word, depending on the number of events in the system (the event queues can be defined to be bytes or words, depending on build options). This number is an index into this table.

(3) The default priority refers to the queue which the event is normally posted to, unless specified otherwise.

(4) The state machine number is an index into the state machine definition table. Each event belongs to (only) one state machine and is effectively an input to that state machine.

(5) The local event number is an index into the state transition table for that state machine, and defines the event number within the state machine to which it belongs.

(6) Event number 0 is a dummy event, which is not included in the table. It is never used as such, but is used to indicate an event which has been cancelled (and hence will not be processed). Therefore, the lowest event in the table is 1 (not 0).

2.5.2 Event Queue Definition Table

(1) This is defined as follows:


```
struct execQueueDef
{
      word or byte NumberOfElements;
      word or byte *QueueStart;
};

struct execQueueDef execQueue[TOTAL_QUEUES];
```

(2) The array `execQueue` is user-supplied, as are the queues themselves.

(3) The event queues are made up of a single byte or word array for all the queues (i.e. the queues are contiguous). The size of this array, `execEventQueue`, is the total of all the elements in all the queues.

(4) It is important that the queue start pointers are set up correctly so that one queue is adjacent to the next, with the highest priority queue at the lowest address (i.e. effectively in reverse order).

(5) The number of queues in the system is given by the constant `TOTAL_QUEUES`.

2.5.3 Input Event Definition Table

(1) An input event definition table is defined as follows:

```
struct execInputEvents
{
      word or byte HighEvent[8];
      word or byte LowEvent [8];
      byte BitMask;
};
```

(2) One of these exists for each byte of input that is required to be processed using the input event posting facility, and the names of these are user-defined. There need be none at all if it is not required to generate events from inputs in this way.

(3) If there are bits in an input byte that need to be processed in a special manner, e.g. at different scan rates, then more than one of these tables can exist for a given input byte. The bit mask determines which bits are relevant.

(4) The high event defines the event number to be posted on a low to high transition, and the low event defines the event number to be posted on a high to low transition.

(5) The first elements of the `High`/`LowEvent` array (i.e. `HighEvent[0]`, `LowEvent[0]`) correspond to the least significant bit of the `BitMask`.

(6) An event number of 0 indicates that no event is to be posted. This is useful if it is only required to post an event for the transition in one direction.

2.5.4 State Machine Definition Table

(1) This is defined as follows:

```
struct execStateMachineDef
{
      byte NumberOfStates;
      byte NumberOfEvents;
      struct execStateTransitionDef *TransitionTable;
};

struct execStateMachineDef execStateMachine[STATE_MACHINES];
```

(2) The array `execStateMachine` is user-supplied, as are the state transition tables themselves (see section 2.5.5 for the definition of `TransitionTable`, which also defines the number of states and events).

(3) The number of state machines in the system is given by the constant `STATE_MACHINES`.

2.5.5 State Transition Table

(1) Each element in a state transition table is defined as follows:

```
struct execStateTransitionDef
{
      byte NextState;
      byte Tag;
      void (*EventFunction)(word Tag);
};
```

(2) A state transition table is a two-dimensional array of state transition elements as follows:

```
execStateTransitionElement MyStateTable[STATES][EVENTS];
```

(3) Therefore, in processing an event, the executive first looks up in the event definition table to find out which state machine that event belongs to, and converts the global event number into a local event. It takes the current state of the appropriate state machine and does a lookup in the state transition table, based on event and the current state, to find out the next state and event function. It calls the event function and, when it returns, it sets the current state of that state machine to the new state as looked up, or if the state transition has been overridden by processing in the event function, to the overridden state.

(4) The `NextState` data item is a number based on 1 rather than 0 (i.e. the first state in the table is state 1 *not* state 0). A zero state value is used to represent a disabled state machine. When a state machine is enabled, it is set to a known starting state, and a state machine may be disabled at any time.

(5) The event function pointer may be null, in which case the state transition is made, but no function is called.

(6) The purpose of `Tag` is to identify the state transition if the same function is called for different transitions. This is placed in the global variable `execTransitionTag` prior to calling the event function.

(7) The state transition tables are user-provided and may be given any names that may be desired. `MyStateTable` is an example in this case.

(8) A valid entry must exist for each event in each state. It is recommended that for 'invalid' state-event combinations, that an exception notification function is called, and a transition is defined to a state that would be most appropriate to cause recovery from the situation.

2.5.6 Structure Definition Array

(1) Each element in a structure definition array is defined as follows:

```
struct execStructDef
{
    word Size;
    word Elements;
    word Offset;
};
```

(2) This structure is used to define the structure of a structure, or simple array or data item, which is to be passed (in a portable way) over a communications link. The pack/unpack facility uses this information (contained in ROM) to convert the given structure into and back out of byte-packed big-endian format.

(3) A particular definition either consists of a single instance of this structure if it describes a simple array or data item, or an array of these structures if a structure or an array of structures is to be described.

(4) A series of macros are provided in order to simplify the construction of descriptor arrays. These are:
(a) `PACK_ELEMENT (structure, type, member, number)`
(b) `PACK_STRUCT (structure, type, member, number)`
(c) `PACK_SINGLE (type, number)`
(d) `PACK_STRUCT_START (structure, number)`
(e) `PACK_STRUCT_END ()`

(5) The use of these macros is best described by example. To describe a simple array, e.g. `word myArray[5];`, use the following:

```
struct execStructDef myArrayDescriptor = PACK_SINGLE (word, 5);
```

(6) When describing structures, any level of nested structure arrays may be used. For the following:

```
struct myStruct1
{
    byte m_S1E1[2];
```

```
    word m_S1E2;
};

struct myStruct2
{
    float m_S2E1;
    struct myStruct1 m_S2E2[3];
    word m_S2E3;
};

struct myStruct3
{
    byte m_S3E1[10];
    struct myStruct1 m_S3E2;
    slword m_S3E3;
    struct myStruct2 m_S3E4[4];
};
```

… the third compound structure is described as follows:

```
struct execStructDef myStruct3Descriptor[16] =
{
  PACK_STRUCT_START (struct myStruct3, 1),
    PACK_ELEMENT (struct myStruct3, byte, m_S3E1, 10),
    PACK_STRUCT (struct myStruct3, struct myStruct1, m_S3E2, 1),
      PACK_ELEMENT (struct myStruct1, byte, m_S1E1, 2),
      PACK_ELEMENT (struct myStruct1, word, m_S1E2, 1),
    PACK_STRUCT_END (),
    PACK_ELEMENT (struct myStruct3, slword, m_S3E3, 1),
    PACK_STRUCT (struct myStruct3, struct myStruct2, m_S3E4, 4),
      PACK_ELEMENT (struct myStruct2, float, m_S2E1, 1),
      PACK_STRUCT (struct myStruct2, struct myStruct1, m_S3E2, 3),
        PACK_ELEMENT (struct myStruct1, byte, m_S1E1, 2),
        PACK_ELEMENT (struct myStruct1, word, m_S1E2, 1),
      PACK_STRUCT_END (),
      PACK_ELEMENT (struct myStruct2, word, m_S2E3, 1),
    PACK_STRUCT_END (),
  PACK_STRUCT_END ()
};
```

(7) Note that the pack and unpack routines use recursion to handle nested structure definitions. On microcontrollers where stack space is very short, then the use of complex nested structures should be avoided, unless the microcontroller is an 8-bit big-endian device (such as the 8051), in which case packing/unpacking is not necessary on the target processor.

2.5.7 Incoming Asynchronous Data Stream State

(1) This block of data is defined as follows:

```
struct execAsyncStateDef
{
    word Checksum;
    enum {OK, Repeat, Overflow, Error, Hunting,
          InitialEscape, Node, Sequence, Receiving,
          EscapeSequence, Checksum1, Checksum2} DecodeState;
```

```
    byte ThisNode;
    byte MessageNode;
    byte DataSize;
    byte ThisSequence;
    byte LastSequence;
};
```

(2) One of these blocks is required for each incoming asynchronous data stream. This data is maintained by the function `execAsyncByteIn()`, with the exception of `ThisNode` which is set up by the application. This function alters the data in this structure so that, next time it is called, it is operating on the same data as previous.

(3) The `Checksum` is a CRC-16 checksum and represents the checksum of the accumulated received data.

(4) The `DecodeState` is used to track the progress of the incoming data. When reception of a message is completed or aborted, the function `execAsyncByteIn()` returns true. When this happens and `DecodeState` is `OK`, then the message should be processed and replied to (on a slave). If `DecodeState` is `Repeat` then the previous reply should be transmitted, unless the message is received while transmitting the previous reply, in which case it is advisable not to reply, so as to allow the master to resynchronise. A `DecodeState` of `Overflow` or `Error` indicates a corrupted incoming message which should not be replied to (since it may be the node number which is corrupted – in which case a reply would be expected from another node). Under these circumstances, the master will almost certainly be expecting a reply and will not get one. However, this information may be useful in gathering statistics about line quality. Note that an overflow occurs if more than 255 bytes of message data are received (not including repeated DLE's), and this may indicate a software problem at the other end of the link.

(5) The `DecodeState` on the master is used similarly to the slave, except that the state is used to determine whether to proceed with a new exchange or to do a retry on the old.

(6) `ThisNode` is used to filter messages so that only messages for a particular node are received. If the node number does not match then the whole message will be ignored and the application will not be informed. `ThisNode` should be set to the node number of the controller in question, or 255 if all messages are to be received. An incoming node number of 255 is treated as broadcast, and the incoming message will be registered regardless of the value of `ThisNode`. A broadcast should not be replied to.

(7) The `MessageNode` is the node number embedded in the incoming message. Normally this will be the same as `ThisNode` when a message has been received. This may not be the case if either the incoming message node or `ThisNode` is set to 255.

(8) The `DataSize` is the size of the data (in bytes) as it is placed in the application-provided buffer (i.e. with leading and trailing data and double DLE's removed). It increments as the data is being received.

(9) `ThisSequence` and `LastSequence` are used in determining whether the message is a repeat or not. Note that after a correctly received message, these will always be the same.

2.5.8 Other Values

(1) The following values, defined in the application header file "ExecApp.h", are required to be provided for the executive to operate correctly:

(a) `TOTAL_QSIZE`
(b) `TOTAL_QUEUES`
(c) `TOTAL_EVENTS`
(d) `TOTAL_TIMERS`
(e) `STATE_MACHINES`

(2) The file "ExecApp.h" is included by "Exec.h" and the values defined are used by the executive when it is compiled. This means that the executive must be freshly compiled with each change to this file, and cannot be distributed as a pre-compiled library.

(3) By convention, "ExecApp.h" contains all the definitions required globally by the application as well as the executive (see the example in section 4.2).

(4) Alternatively, the application header file may be given a different name, such as "MyApp.h", in which case the constant APP_HEADER must be defined as "MyApp.h" in the compile options, typically appearing as APP_HEADER="\"MyApp.h\"".

2.5.9 Generating Definitions Automatically

(1) The various data structures and definitions contain numerous cross-dependencies which, if broken, will cause an application or, worse, part of it, to malfunction. There is therefore a script-based facility to automatically insert the relevant code into the appropriate source code files.

(2) This facility uses the 'Python' scripting language. Python is free software and can be downloaded from the Internet at www.python.org. It comes complete with Python interpreter (for Windows, most flavours of Unix and Macintosh), library and comprehensive documentation. It is very easy for 'C' programmers to learn, and the facility supplied with the State-Event Executive can easily be expanded to generate other parts of application code (e.g. to create string tables to give event trace data meaning).

(3) The automatically-generated parts of the code are contained in special comment-delimited sections. The pattern of the delimiting is as follows: the start of a section begins with `//{{XXX(Y)`, where `XXX(Y)` is a pseudo-macro distinguishing one section from another. The section ends with `//}}`. The script generates these

sections internally and then searches all the specified source files, and matches these sections in the existing files against the sections it has generated, substituting in any changes. Thus, these sections must already pre-exist in the source files before the tool is run (though they can be empty, ready for the tool to insert the real code). Note that the tool only modifies the files with changes, and warns of any sections that have not been found or appear in more than one place.

(4) The facility is driven from a main script which is defined by the application. This scripts defines all the necessary table data and imports/calls functions in 'ExecStuff.py' which actually manipulates the data, stuffing the generated code into the source files. Running the application-defined script with Python from the command line results in the source files being updated, and a log being output to the screen indicating what has been done. The easiest way to understand how the application defined script is constructed is to examine and then modify the example script (see section 4.1). Note that any errors will manifest themselves as Python exceptions, which will abort the script.

(5) The following is a complete list of the pseudo-macros used by ExecStuff.py.

(a) Used by the function `genStateTables()`:

(i) `EXEC_DEFINE()` – this contains all the manifest constants and declarations in the main application header file.

(ii) `EXEC_DEFINITION()` – this contains the overall definition of the set of state machines

(iii) `EXEC_DECLARE()` – this contains the declarations of all the state machines (for the appropriate header file).

(iv) `EXEC_SM_DECLARE(`*`sm`*`)`, where *`sm`* is the manifest constant for a particular state machine – this contains the state declarations for a given state machine.

(v) `EXEC_SM_DEFINE(`*`sm`*`)`, where *`sm`* is the manifest constant for a particular state machine – this contains the state definition table for a given state machine.

(b) Used by the function `genTimers()`:

(i) `EXEC_TIMERS()` – this contains the manifest constants of the timers.

(c) Used by the function `genCommsStructures()`:

(i) `EXEC_COMMS_STRUCT(`*`st`*`)`, where *`st`* is the structure tag of the structure being defined – this contains the structure declaration for a given structure.

(ii) `EXEC_COMMS_DESCRIPTOR(`*`st`*`)`, where *`st`* is the structure tag of the structure being defined – this contains the communications packing descriptor for the given structure.

(6) The function `genFiles()` within ExecStuff.py takes a list o fall these sections and does the pattern matching and file stuffing.

(7) The function `genStabReport()` within ExecStuff.py generates an HTML file with the state tables in a more readable tabular form.

3. DESIGN AND IMPLEMENTATION DETAILS OF THE EXECUTIVE

3.1 Posting Events

(1) Posting events is done by the function `execPostPriorityEvent` (execPostEvent is a macro which calls `execPostPriorityEvent` with the default priority for the event).

(2) This must work whether it is called from a state transition function or an interrupt, including interrupts of higher priority that have interrupted other interrupt routines. The posting function therefore has to lock out access to the event queue (by suspending interrupts) during the critical parts of this operation.

(3) In order to prevent the "event retrieving" part of the executive from having to do excessive searching, the posting routine also sets up a request to switch to a higher priority, if necessary. This action also requires a resource lock.

(4) Hence the following global variables are maintained:

```
byte execRequestedLevel;      // Requested priority level

byte execCurrentLevel;        // Current priority level

byte execCurrentStateMachine;

byte execNextState;
```

3.2 Retrieving Events

(1) Retrieving events is done by polling the queues in order of priority, starting with the level requested by the event posting routines. If there are no outstanding events at that level, then the priority moves down one level, and the same exercise is repeated until there is nothing outstanding, at which point the executive enters into idling until an event is posted.

(2) The sequence of processing an event is as follows:

(a) Read the event number, and use it to look up the state machine and local event number.

(b) Set the (global) state machine number to the retrieved value.

(c) If the current state number is zero (disabled), do not process this event, just call the trace function and remove the event from the queue.

(d) Otherwise, from the state machine and local event numbers, find the state transition.

(e) Set the (global) next state value from the number in the state transition.

(f) Call the state transition function (if there is one), passing the appropriate variables.

(g) Call the user's trace function with the appropriate parameters.

(h) If the current state is not zero (i.e. disabled), set the current state of the state machine to the next state value.

(3) Note that the event queue needs to be locked when retrieving an event.

(4) Note also that the general rule is that state machines should manipulate themselves, primarily by making the transitions defined in the state transition tables in response to events. There is provision to override default actions, however.

3.3 Handling Timers

(1) The timer functionality requires an interface to a (16-bit) timer hardware resource, and this interface is provided by the user (application).

(2) The intended implementation is that the timer should be set up to run for a given length of time before it issues an interrupt, and which point the timer is set to run for a new length of time, or disabled if the timer function is not in use (for the time being). It is intended that the timer is a continuous free-running timer which wraps round at FFFF. A compare register can then be set up to whatever value is required.

(3) An alternative implementation involves the use of a timer tick interrupt. In this case the timer value is maintained in software, and the timer comparison is also done in software. This kind of implementation is typically used when the scanning of inputs coincides with the timer tick.

(4) The main timer may count up, or it may optionally count down (define COUNTDOWN when compiling).

3.4 Internal Data Structures

3.4.1 Event Trace Enable Table

(1) This is defined as follows:

```
byte execTraceTable[(TOTAL_EVENTS/8) + 1];
```

(2) This is actually a table of bits where a '1' corresponds to tracing on that event number to be enabled. Bit 0 is a global trace enable bit and this must be 1 for tracing on any of the other events to be performed.

(3) This table is located in RAM and is controlled by debug/diagnostic facilities in the application.

(4) When an event occurs and tracing for that event is enabled, the user defined function `execAppTrace` is called (see section 2.2.1.7).

3.4.2 Current State Table

(1) This is defined as follows:

```
byte execCurrentState[STATE_MACHINES];
```

(2) This is defined in RAM and is used by the executive to maintain the current state of each state machine.

3.4.3 Timer Queue

(1) This is defined as follows (and stored in RAM):

```
struct execTimerDef
{
      word TriggerPoint;
      word or byte EventToPost;
      word or byte PreviousNumber;
      word or byte NextNumber;
};

struct execTimerDef execTimer[TOTAL_TIMERS];
```

(2) All events are single shot, i.e. they disable themselves once they have triggered.

(3) The timer system is implemented using a single (word-sized) main timer which counts up (or optionally down). When a timer is started, the time delay value is added to the main timer value and stored as the trigger point. The event is then posted when the main timer reaches the trigger point.

(4) The timer system is implemented using a doubly-linked-list queuing mechanism, chronologically ordered. Setting a timer adds an entry to the queue, and it is removed either when the timer times out or when it is killed.

3.4.4 Data Queues

(1) A data queue is defined as follows (and stored in RAM):

```
struct execDataQueue
{
      word or byte Tail;
```



```
        word or byte Count;
        byte Data[];
};
```

(2) Each element of data in a queue is preceded by a size byte, which contains the number of bytes in the data element which follows it.

(3) The data part of each queue is a circular queue, able to contain variable length elements. Thus buffers are copied in and out of a queue and cannot be accessed directly.

(4) Thus, a data queue is effectively a chain of buffers as follows:

Tail	Count	1st size	1st buffer …	2nd size	2nd buffer …	3rd size	3rd buffer …	…

4. APPLICATION EXAMPLE

4.1 Script for Generating Source Code Tables

The following Python script, Widget.py, will generate the declarations and tables for the example in the rest of this section, but the code examples that follow have not been generated using this facility.

```
import sys
sys.path.append("..\\exec")  # for ExecStuff, assuming exec is in this directory
from ExecStuff import *

############# Complete set of definitions for exec, etc. ######################

Definition = {
    'Event Queues' : (('BACKGND_QUEUE', 128),  ('LOW_QUEUE', 32),
                      ('INTERMED_QUEUE', 32), ('TOP_QUEUE', 8) ),

    'State Tables' : (
        ( 'smSingleTransition',
         ('SM_SINGLE_STATE',                 'STATE1'),
#          ========================================================================
        (('EV_STATUS_REQUEST', 'BACKGND_QUEUE'),
                                           ('STATE1', 0, 'HandleStatusRequest')),
        (('EV_SCAN_TRIGGER',   'TOP_QUEUE'), ('STATE1', 0, 'ScanInputs'))
        ),

        ( 'smWidgetTransition',
         ('SM_WIDGET_SEQUENCER', 'WIDGET_IDLE', 'ENTERING_SLOT', 'LEAVING_SLOT',
                                 'ENTERING_TRAY', 'AWAITING_REMOVAL', 'FAULTY'),
#          ========================================================================
            (('EV_DOOR_READY',     'INTERMED_QUEUE'),
                    ('ENTERING_SLOT',  0, 'DispenseWidget'), # WIDGET_IDLE
                    ('ENTERING_SLOT',  0, None),             # ENTERING_SLOT
                    ('LEAVING_SLOT',   0, None),             # LEAVING_SLOT
                    ('ENTERING_TRAY',  0, None),             # ENTERING_TRAY
                    ('ENTERING_SLOT',  0, 'DispenseWidget'), # AWAITING_REMOVAL
                    ('FAULTY',         0, None)),            # FAULTY

            (('EV_WIDGET_IN_SLOT', 'INTERMED_QUEUE'),
                    ('FAULTY',         1, 'LogWidgetFault'), # WIDGET_IDLE
                    ('LEAVING_SLOT',   0, None),             # ENTERING_SLOT
                    ('LEAVING_SLOT',   0, None),             # LEAVING_SLOT
                    ('FAULTY',         3, 'LogWidgetFault'), # ENTERING_TRAY
                    ('FAULTY',         3, 'LogWidgetFault'), # AWAITING_REMOVAL
                    ('FAULTY',         0, None)),            # FAULTY
            (('EV_SLOT_CLEAR',     'INTERMED_QUEUE'),
                    ('WIDGET_IDLE',    0, None),             # WIDGET_IDLE
                    ('ENTERING_SLOT',  0, None),             # ENTERING_SLOT
                    ('ENTERING_TRAY',  0, 'HandleWidget'),   # LEAVING_SLOT
                    ('ENTERING_TRAY',  0, None),             # ENTERING_TRAY
                    ('AWAITING_REMOVAL',0,None),             # AWAITING_REMOVAL
                    ('FAULTY',         0, None)),            # FAULTY
            (('EV_WIDGET_IN_TRAY', 'INTERMED_QUEUE'),
                    ('FAULTY',         2, 'LogWidgetFault'), # WIDGET_IDLE
                    ('FAULTY',         2, 'LogWidgetFault'), # ENTERING_SLOT
                    ('AWAITING_REMOVAL',0,'StopWidget'),     # LEAVING_SLOT
                    ('AWAITING_REMOVAL',0,'StopWidget'),     # ENTERING_TRAY
                    ('AWAITING_REMOVAL',0,None),             # AWAITING_REMOVAL
                    ('FAULTY',         0, None)),            # FAULTY
            (('EV_TRAY_CLEAR',     'INTERMED_QUEUE'),
                    ('WIDGET_IDLE',    0, None),             # WIDGET_IDLE
                    ('ENTERING_SLOT',  0, None),             # ENTERING_SLOT
                    ('LEAVING_SLOT',   0, None),             # LEAVING_SLOT
                    ('WIDGET_IDLE',    0, 'RecordWidget'),   # ENTERING_TRAY
                    ('WIDGET_IDLE',    0, 'RecordWidget' ),  # AWAITING_REMOVAL
                    ('FAULTY',         0, None)),            # FAULTY
            (('EV_WIDGET_TIMEOUT', 'INTERMED_QUEUE'),
                    ('WIDGET_IDLE',    0, None),             # WIDGET_IDLE
                    ('FAULTY',        10, 'LogWidgetFault'), # ENTERING_SLOT
                    ('FAULTY',        10, 'LogWidgetFault'), # LEAVING_SLOT
```

```
                        ('FAULTY',        10, 'LogWidgetFault'), # ENTERING_TRAY
                        ('WIDGET_IDLE',    0, 'RecordWidget'),   # AWAITING_REMOVAL
                        ('WIDGET_IDLE',    0, None))             # FAULTY
        ),

        ( 'smDoorTransition',
         ('SM_DOOR_CONTROL', 'DOOR_CLOSED', 'DOOR_OPENING',
                             'DOOR_OPEN', 'DOOR_CLOSING'),
#          ================================================================
             (('EV_BUTTON_PUSHED', 'LOW_QUEUE'),
                        ('DOOR_OPENING',   0, 'OpenDoor'),       # DOOR_CLOSED
                        ('DOOR_OPENING',   0, None),             # DOOR_OPENING
                        ('DOOR_OPEN',      0, None),             # DOOR_OPEN
                        ('DOOR_OPENING',   0, 'OpenDoor')),      # FAULTY
             (('EV_DOOR_OPEN',     'LOW_QUEUE'),
                        ('DOOR_CLOSED',    0, None),             # DOOR_CLOSED
                        ('DOOR_OPEN',      0, 'TriggerWidget'),  # DOOR_OPENING
                        ('DOOR_OPEN',      0, None),             # DOOR_OPEN
                        ('DOOR_CLOSING',   0, None)),            # FAULTY
             (('EV_DOOR_CLOSED',   'LOW_QUEUE'),
                        ('DOOR_CLOSED',    0, None),             # DOOR_CLOSED
                        ('DOOR_OPENING',   0, None),             # DOOR_OPENING
                        ('DOOR_OPEN',      0, None),             # DOOR_OPEN
                        ('DOOR_CLOSING',   0, None)),            # FAULTY
             (('EV_DOOR_TIMEOUT',  'LOW_QUEUE'),
                        ('DOOR_CLOSED',    0, None),             # DOOR_CLOSED
                        ('DOOR_CLOSING',   0, 'CloseDoor'),      # DOOR_OPENING
                        ('DOOR_CLOSING',   0, 'CloseDoor'),      # DOOR_OPEN
                        ('DOOR_CLOSING',   0, None))             # FAULTY
        ),
    ),

    'Timers' : ('WIDGET_TIMER', 'DOOR_TIMER'),

    'Comms' : (('boolean', 'byte', 'sbyte', 'word', 'sword', 'lword', 'slword'),

               (('StatusDef', 'StatusDefDescriptor', 1),
                ('byte', 'ID', 1),
                ('lword', 'Flags', 2),
                ('DateTimeDef', 'Timestamp', 1)),

               (('DateTimeDef', None, 0),
                ('byte', 'Hour', 1),
                ('byte', 'Minute', 1),
                ('byte', 'Second', 1),
                ('byte', 'Day', 1),
                ('byte', 'Month', 1),
                ('word', 'Year', 1))
              ),

    'Source Files' : ('Widget.h', 'Widget_d.h', 'Widget00.c', 'Widget01.c',
                                  'Widget02.c', 'Widget03.c', 'Widget04.c')
}

################################################################################

genFiles( genStateTables (Definition['Event Queues'], Definition['State Tables']) +
          genTimers (Definition['Timers']) +
          genCommsStructures (Definition['Comms']),
          Definition['Source Files'])

genStabReport( Definition['State Tables'], 'WidgetStates')

print 'Source code update complete.'
```

4.2 Application Definition Header File Data

The following is a section from Widget_d.h (which included by Exec.h for use in the executive and the application, by defining the constant APP_HEADER to be “Widget_d.h” in the compile options – this typically appears as APP_HEADER="\"Widget_d.h\"" in the command line):

```
#ifdef __C166__
#include "C167Dep.h"
#else
#include "WinDep.h"
#endif

//{{EXEC_DEFINE() – if used, this would go here and be filled with the following:
//}}

/*********************  Queues  *********************************************/

#define BACKGND_QSIZE   128
#define LOW_QSIZE        32
#define INTERMED_QSIZE   32
#define TOP_QSIZE         8
#define TOTAL_QSIZE    (BACKGND_QSIZE + LOW_QSIZE + INTERMED_QSIZE + TOP_QSIZE)

#define BACKGND_QUEUE     0
#define LOW_QUEUE         1
#define INTERMED_QUEUE    2
#define TOP_QUEUE         3
#define TOTAL_QUEUES      4

/*********************  Events  *********************************************/

#define NON_EVENT                0
#define EV_STATUS_REQUEST        1
#define EV_SCAN_TRIGGER          2
#define EV_DOOR_READY            3
#define EV_WIDGET_IN_SLOT        4
#define EV_SLOT_CLEAR            5
#define EV_WIDGET_IN_TRAY        6
#define EV_TRAY_CLEAR            7
#define EV_WIDGET_TIMEOUT        8
#define EV_BUTTON_PUSHED         9
#define EV_DOOR_OPEN            10
#define EV_DOOR_CLOSED          11
#define EV_DOOR_TIMEOUT         12

#define TOTAL_EVENTS            12

/*********************  State Machines  *************************************/

#define SM_SINGLE_STATE          0
#define SM_WIDGET_SEQUENCER      1
#define SM_DOOR_CONTROL          2

#define STATE_MACHINES           3

#define SINGLE_EVENTS            2
#define WIDGET_STATES            6
#define WIDGET_EVENTS            6
#define DOOR_STATES              4
#define DOOR_EVENTS              4

/*********************  Timers  *********************************************/

//{{EXEC_TIMERS() – if used, this would go here and be filled with the following:
//}}

#define WIDGET_TIMER             0
#define DOOR_TIMER               1

#define TOTAL_TIMERS             2

#define WIDGET_TIME              2000
#define DOOR_TIME                 500
```

4.3 Application Main Header File Data

The following is a section from Widget.h:

```
extern VCONST struct execInputEventDef InputEvents;
```

```
//{{EXEC_DECLARE() – if used, this would go here and be filled with:
//}}

extern VCONST struct execStateTransitionDef smSingleTransition[SINGLE_EVENTS];
extern VCONST struct execStateTransitionDef
                            smWidgetTransition[WIDGET_STATES*WIDGET_EVENTS];
extern VCONST struct execStateTransitionDef
                            smDoorTransition[DOOR_STATES*DOOR_EVENTS];
```

4.4 Application-Provided Data Structures

The following is a section from Widget00.c (module 00 contains the definition tables by convention):

```
#include "Exec.h"

//{{EXEC_DEFINITION() – if used, this would go here and be filled with:
//}}

// Event queue definitions
// -----------------------
VCONST struct execQueueDef execQueue[TOTAL_QUEUES] =
{
    {BACKGND_QSIZE ,&execEventQueue[0]},
    {LOW_QSIZE,&execEventQueue[BACKGND_QSIZE]},
    {INTERMED_QSIZE,&execEventQueue[BACKGND_QSIZE+LOW_QSIZE]},
    {TOP_QSIZE,&execEventQueue[BACKGND_QSIZE+LOW_QSIZE+INTERMED_QSIZE]}
};

// State machine definitions
// -------------------------
VCONST struct execStateMachineDef execStateMachine[STATE_MACHINES] =
{
    {1,                SINGLE_EVENTS,    smSingleTransition},
    {WIDGET_STATES,    WIDGET_EVENTS,    smWidgetTransition},
    {DOOR_STATES,      DOOR_EVENTS,      smDoorTransition}
};

// Event definitions
// -----------------
VCONST struct execEventDef execEvent[TOTAL_EVENTS+1] =
{
    {0,              0, 0 },                    // NON_EVENT
    {BACKGND_QUEUE,  0, SM_SINGLE_STATE},       // EV_STATUS_REQUEST
    {TOP_QUEUE,      1, SM_SINGLE_STATE},       // EV_SCAN_TRIGGER
    {INTERMED_QUEUE, 0, SM_WIDGET_SEQUENCER},   // EV_DOOR_READY
    {INTERMED_QUEUE, 1, SM_WIDGET_SEQUENCER},   // EV_WIDGET_IN_SLOT
    {INTERMED_QUEUE, 2, SM_WIDGET_SEQUENCER},   // EV_SLOT_CLEAR
    {INTERMED_QUEUE, 3, SM_WIDGET_SEQUENCER},   // EV_WIDGET_IN_TRAY
    {INTERMED_QUEUE, 4, SM_WIDGET_SEQUENCER},   // EV_TRAY_CLEAR
    {INTERMED_QUEUE, 5, SM_WIDGET_SEQUENCER},   // EV_WIDGET_TIMEOUT
    {LOW_QUEUE,      0, SM_DOOR_CONTROL},       // EV_BUTTON_PUSHED
    {LOW_QUEUE,      1, SM_DOOR_CONTROL},       // EV_DOOR_OPEN
    {LOW_QUEUE,      2, SM_DOOR_CONTROL},       // EV_DOOR_CLOSED
    {LOW_QUEUE,      3, SM_DOOR_CONTROL}        // EV_DOOR_TIMEOUT
};

// Input event definitions
// -----------------------
VCONST struct execInputEventDef InputEvents  =
{
    {EV_BUTTON_PUSHED, NON_EVENT,         NON_EVENT,         EV_DOOR_OPEN,
     NON_EVENT,        EV_WIDGET_IN_SLOT, EV_WIDGET_IN_TRAY, NON_EVENT},

    {NON_EVENT,        NON_EVENT,         NON_EVENT,         EV_DOOR_CLOSED,
     NON_EVENT,        EV_SLOT_CLEAR,     EV_TRAY_CLEAR,     NON_EVENT},

    0x96        // Bit mask; high-going events (above), low-going events
};
```

4.5 Application-Provided Routines

The following is a section from Widget01.c (module 01 contains the application-provided exec. functions by convention):

```
#include <stddef.h>
#include "Exec.h"

void execAppInit (void)
{
//  Perform integrity checks
//  ------------------------
    if (execCheckIntegrity (0) != 0)
    {
        IdleMonitorPin = FALSE;        // Signal system as non-starter
        while (1);                     // Halt
    }
    else
        IdleMonitorPin = TRUE;

//  Initialise the system
//  ---------------------
    SetupIOandInterrupts();
    execEnableStateMachine (SM_SINGLE_STATE, 1);
    execEnableStateMachine (SM_WIDGET_SEQUENCER, 1);
    execEnableStateMachine (SM_DOOR_CONTROL, 1);
}

void execAppIdle(void)
{
    IdleMonitorPin = FALSE;
    execSafeCpuIdle();      // Returns after next interrupt has processed
    IdleMonitorPin = TRUE;
}

void execAppTrace (byte BeforeState, byte AfterState, word Event)
{
//  Assumes maximum of 15 states and 64 events
//  ------------------------------------------
    EmitCodeSomewhere ( (word)(((BeforeState&15)<< 10) +
                               ((AfterState&15)<< 6) +
                                Event&63));
}

#ifndef _WIN32

void TimerInterrupt (void) /* interrupt */
{
    word NextTick, CurrentTick = HwareTimerReg;

//  Process the timeout(s)
//  ----------------------
    NextTick = execProcessTimeouts(CurrentTick);

//  Set up for next interrupt or disable
//  ------------------------------------
    if (NextTick == CurrentTick)
        HwareTimerEnable = FALSE;          // Disable timer compare
    else
        HwareCompareReg = NextTick;
}

word execAppCurrentTick()
{
//  Return current timer tick
//  -------------------------
    return HwareTimerReg;
}

void execAppResyncTimer (word TickCount, boolean Enable)
{
    HwareTimerEnable = FALSE;          // Disable timer compare

//  Set up for next interrupt
```

```
//  ------------------------
    HwareCompareReg = TickCount;

//  Enable timer compare if required
//  --------------------------------
    if (Enable);
        HwareTimerEnable = TRUE;        // Enable timer compare
}

#else

void execAppFinish()
{
}

#endif  // _WIN32
```

4.6 State Machines

The following sections are from Widget02.c, Widget03.c, etc., and each contains one state machine each. Module 02 contains the single-state state machine by convention, which is a collection of all of the event handling functions for which state information is irrelevant.

Widget02.c:

```
#include <stddef.h>
#include "Exec.h"

/*********************  State transition table *******************************/

//{{EXEC_SM_DECLARE(SM_SINGLE_STATE) – if used, this would go here, filled with:
//}}

//  States
//  ------
#define STATE1  1       // The only valid state

//  State transition functions
//  --------------------------
static void HandleStatusRequest(void);
static void ScanInputs(void);

//{{EXEC_SM_DEFINE(SM_SINGLE_STATE) – if used, this would go here, filled with:
//}}

//  State table
//  -----------
VCONST struct execStateTransitionDef smSingleTransition[SINGLE_EVENTS] =
{
//  STATE1:
//  ------
    {STATE1,    0, HandleStatusRequest},    // EV_STATUS_REQUEST
    {STATE1,    0, ScanInputs}              // EV_SCAN_TRIGGER
};

/*********************  State transition functions ***************************/

//  Handle status request
//  ---------------------
static void HandleStatusRequest (void)
{
    ReplyToRequest();
}

//  Trigger events based on input pin transitions
//  ---------------------------------------------
static void ScanInputs (void)
{
    static byte OldInputs = 0x0000, ValidatedInputs = 0x0000;
```

```
    execPostInputEvents (PinPort, &OldInputs, &ValidatedInputs, &InputEvents);
}
```

Widget03.c; Widget sequencer state machine:

```
#include <stddef.h>
#include "Exec.h"

/*********************  State transition table ******************************/

//{{EXEC_SM_DECLARE(SM_WIDGET_SEQUENCER) – if used, this would go here, with:
//}}

//  States
//  ------
#define WIDGET_IDLE       1
#define ENTERING_SLOT     2
#define LEAVING_SLOT      3
#define ENTERING_TRAY     4
#define AWAITING_REMOVAL  5
#define FAULTY            6


//  State transition functions
//  --------------------------
static void DispenseWidget(void);
static void HandleWidget(void);
static void StopWidget(void);
static void RecordWidget(void);
static void LogWidgetFault(void);

//{{EXEC_SM_DEFINE(SM_WIDGET_SEQUENCER) – if used, this would go here, with:
//}}

//  State table
//  -----------
VCONST struct execStateTransitionDef smWidgetTransition[WIDGET_STATES *
                                                        WIDGET_EVENTS] =
{
//  WIDGET_IDLE:
//  -----------
    {ENTERING_SLOT,  0, DispenseWidget},    // EV_DOOR_READY
    {FAULTY,         1, LogWidgetFault},    // EV_WIDGET_IN_SLOT
    {WIDGET_IDLE,    0, NOFUNC},            // EV_SLOT_CLEAR
    {FAULTY,         2, LogWidgetFault},    // EV_WIDGET_IN_TRAY
    {WIDGET_IDLE,    0, NOFUNC},            // EV_TRAY_CLEAR
    {WIDGET_IDLE,    0, NOFUNC},            // EV_WIDGET_TIMEOUT

//  ENTERING_SLOT:
//  -------------
    {ENTERING_SLOT,  0, NOFUNC},            // EV_DOOR_READY
    {LEAVING_SLOT,   0, NOFUNC},            // EV_WIDGET_IN_SLOT
    {ENTERING_SLOT,  0, NOFUNC},            // EV_SLOT_CLEAR
    {FAULTY,         2, LogWidgetFault},    // EV_WIDGET_IN_TRAY
    {ENTERING_SLOT,  0, NOFUNC},            // EV_TRAY_CLEAR
    {FAULTY,        10, LogWidgetFault},    // EV_WIDGET_TIMEOUT

//  LEAVING_SLOT:
//  ------------
    {LEAVING_SLOT,   0, NOFUNC},            // EV_DOOR_READY
    {LEAVING_SLOT,   0, NOFUNC},            // EV_WIDGET_IN_SLOT
    {ENTERING_TRAY,  0, HandleWidget},      // EV_SLOT_CLEAR
    {AWAITING_REMOVAL,0,StopWidget},        // EV_WIDGET_IN_TRAY
    {LEAVING_SLOT,   0, NOFUNC},            // EV_TRAY_CLEAR
    {FAULTY,        10, LogWidgetFault},    // EV_WIDGET_TIMEOUT

//  ENTERING_TRAY:
//  -------------
    {ENTERING_TRAY,  0, NOFUNC},            // EV_DOOR_READY
    {FAULTY,         3, LogWidgetFault},    // EV_WIDGET_IN_SLOT
    {ENTERING_TRAY,  0, NOFUNC},            // EV_SLOT_CLEAR
    {AWAITING_REMOVAL,0,StopWidget},      // EV_WIDGET_IN_TRAY
    {WIDGET_IDLE,    0, RecordWidget},    // EV_TRAY_CLEAR
```

```
    {FAULTY,        10, LogWidgetFault},  // EV_WIDGET_TIMEOUT

//  AWAITING_REMOVAL:
//  ----------------
    {ENTERING_SLOT,  0, DispenseWidget},    // EV_DOOR_READY
    {FAULTY,         3, LogWidgetFault},    // EV_WIDGET_IN_SLOT
    {AWAITING_REMOVAL,0,NOFUNC},            // EV_SLOT_CLEAR
    {AWAITING_REMOVAL,0,NOFUNC},            // EV_WIDGET_IN_TRAY
    {WIDGET_IDLE,    0, RecordWidget },     // EV_TRAY_CLEAR
    {WIDGET_IDLE,    0, RecordWidget },     // EV_WIDGET_TIMEOUT

//  FAULTY:
//  ------
    {FAULTY,         0, NOFUNC},            // EV_DOOR_READY
    {FAULTY,         0, NOFUNC},            // EV_WIDGET_IN_SLOT
    {FAULTY,         0, NOFUNC},            // EV_SLOT_CLEAR
    {FAULTY,         0, NOFUNC},            // EV_WIDGET_IN_TRAY
    {FAULTY,         0, NOFUNC},            // EV_TRAY_CLEAR
    {WIDGET_IDLE,    0, NOFUNC}             // EV_WIDGET_TIMEOUT
};

/*********************  State transition functions *************************/

// Start by turning widget dispenser on
// -----------------------------------
static void DispenseWidget(void)
{
    WidgetDispenser (TRUE);
    execSetTimer (WIDGET_TIMER, WIDGET_TIME, EV_WIDGET_TIMEOUT);
}

// Turn widget dispenser off, widget handler on
// -------------------------------------------
static void HandleWidget(void)
{
    WidgetDispenser (FALSE);
    WidgetHandler (TRUE);
    execSetTimer (WIDGET_TIMER, WIDGET_TIME, EV_WIDGET_TIMEOUT);
}

// Turn widget handler off
// ----------------------
static void StopWidget(void)
{
    WidgetHandler (FALSE);
    execSetTimer (WIDGET_TIMER, WIDGET_TIME, EV_WIDGET_TIMEOUT);
}

// Record the removal of a widget by sending a number
// -------------------------------------------------
static void RecordWidget(void)
{
    EmitCodeSomewhere (0x4000 + execTransitionTag);
    execSetTimer (WIDGET_TIMER, WIDGET_TIME, EV_WIDGET_TIMEOUT);
}

// Log fault by sending the fault number somewhere
// ----------------------------------------------
static void LogWidgetFault(void)
{
    EmitCodeSomewhere (0x8000 + execTransitionTag);
}
```

Widget04.c; Door sequencer state machine:

```
#include <stddef.h>
#include "Exec.h"

/*********************  State transition table *****************************/

//{{EXEC_SM_DECLARE(SM_DOOR_CONTROL) – if used, this would go here, with:
//}}

//  States
```

```
//  ------
#define DOOR_CLOSED       1
#define DOOR_OPENING      2
#define DOOR_OPEN         3
#define DOOR_CLOSING      4

//  State transition functions
//  --------------------------
static void OpenDoor(void);
static void TriggerWidget(void);
static void CloseDoor(void);

//{{EXEC_SM_DEFINE(SM_DOOR_CONTROL) – if used, this would go here, with:
//}}

//  State table
//  -----------
VCONST struct execStateTransitionDef smDoorTransition[DOOR_STATES *
                                                      DOOR_EVENTS] =
{
//  DOOR_CLOSED:
//  -----------
    {DOOR_OPENING,   0, OpenDoor},          // EV_BUTTON_PUSHED
    {DOOR_CLOSED,    0, NOFUNC},            // EV_DOOR_OPEN
    {DOOR_CLOSED,    0, NOFUNC},            // EV_DOOR_CLOSED
    {DOOR_CLOSED,    0, NOFUNC},            // EV_DOOR_TIMEOUT

//  DOOR_OPENING:
//  ------------
    {DOOR_OPENING,   0, NOFUNC},            // EV_BUTTON_PUSHED
    {DOOR_OPEN,      0, TriggerWidget},     // EV_DOOR_OPEN
    {DOOR_OPENING,   0, NOFUNC},            // EV_DOOR_CLOSED
    {DOOR_CLOSING,   0, CloseDoor},         // EV_DOOR_TIMEOUT

//  DOOR_OPEN:
//  ---------
    {DOOR_OPEN,      0, NOFUNC},            // EV_BUTTON_PUSHED
    {DOOR_OPEN,      0, NOFUNC},            // EV_DOOR_OPEN
    {DOOR_OPEN,      0, NOFUNC},            // EV_DOOR_CLOSED
    {DOOR_CLOSING,   0, CloseDoor},         // EV_DOOR_TIMEOUT

//  DOOR_CLOSING:
//  ------------
    {DOOR_OPENING,   0, OpenDoor },         // EV_BUTTON_PUSHED
    {DOOR_CLOSING,   0, NOFUNC},            // EV_DOOR_OPEN
    {DOOR_CLOSING,   0, NOFUNC},            // EV_DOOR_CLOSED
    {DOOR_CLOSING,   0, NOFUNC}             // EV_DOOR_TIMEOUT
};

/*********************  State transition functions **************************/

// Activate the door opener to open it
// -----------------------------------
static void OpenDoor(void)
{
    EnergiseDoor (TRUE);
    execSetTimer (DOOR_TIMER, DOOR_TIME, EV_DOOR_TIMEOUT);
}

// Start the widget sequencer off
// ------------------------------
static void TriggerWidget(void)
{
    execPostEvent (EV_DOOR_READY);
}

// De-activate the door opener to close it
// ---------------------------------------
static void CloseDoor(void)
{
    EnergiseDoor (FALSE);
}
```