

IMPLEMENTING THE ACTIVE FRAMEWORK

Iain McInnes 2004-07-27

1.1 WHY YOU MIGHT BE INTERESTED

You might take away any of the following.

- A feel for the flavour of event driven systems
- Decent implementations of timers, event queues, and state machines - all can be used independently.
- Idioms for allocation and initialisation
- Idioms for scheduling and re-entrancy
- Idioms for doing objects in C
- Idioms for partitioning into reusable and custom parts
- Opportunity to read decent code.
- A unit test framework.

1.2 TERMINOLOGY

Active Object:

UML: The root of a thread.

Concurrent Object:

Iain McInnes: A concurrent object is able to execute concurrently with other concurrent objects. Objects may send messages to concurrent objects, where they are queued at the receiver until they can be processed. The concurrent object is the unit of scheduling. A concurrent object may or may not occupy its own thread.

An important class of concurrent objects are those that run to completion: They only block when they have no messages to process.

1.3 TEST DRIVEN DEVELOPMENT

Kent Beck said:

- Use a test harness: Tests should execute automatically.
- Always write test cases before writing the code.
- Execute the test, and be sure the test cases fail.
- Write the simplest code that allows the test cases to pass.
- Develop the code in small increments, and run the tests often.
- Refactor as necessary to keep a clean design.

1.4 TEST HARNESS

Active framework + unit test framework + test harness means:

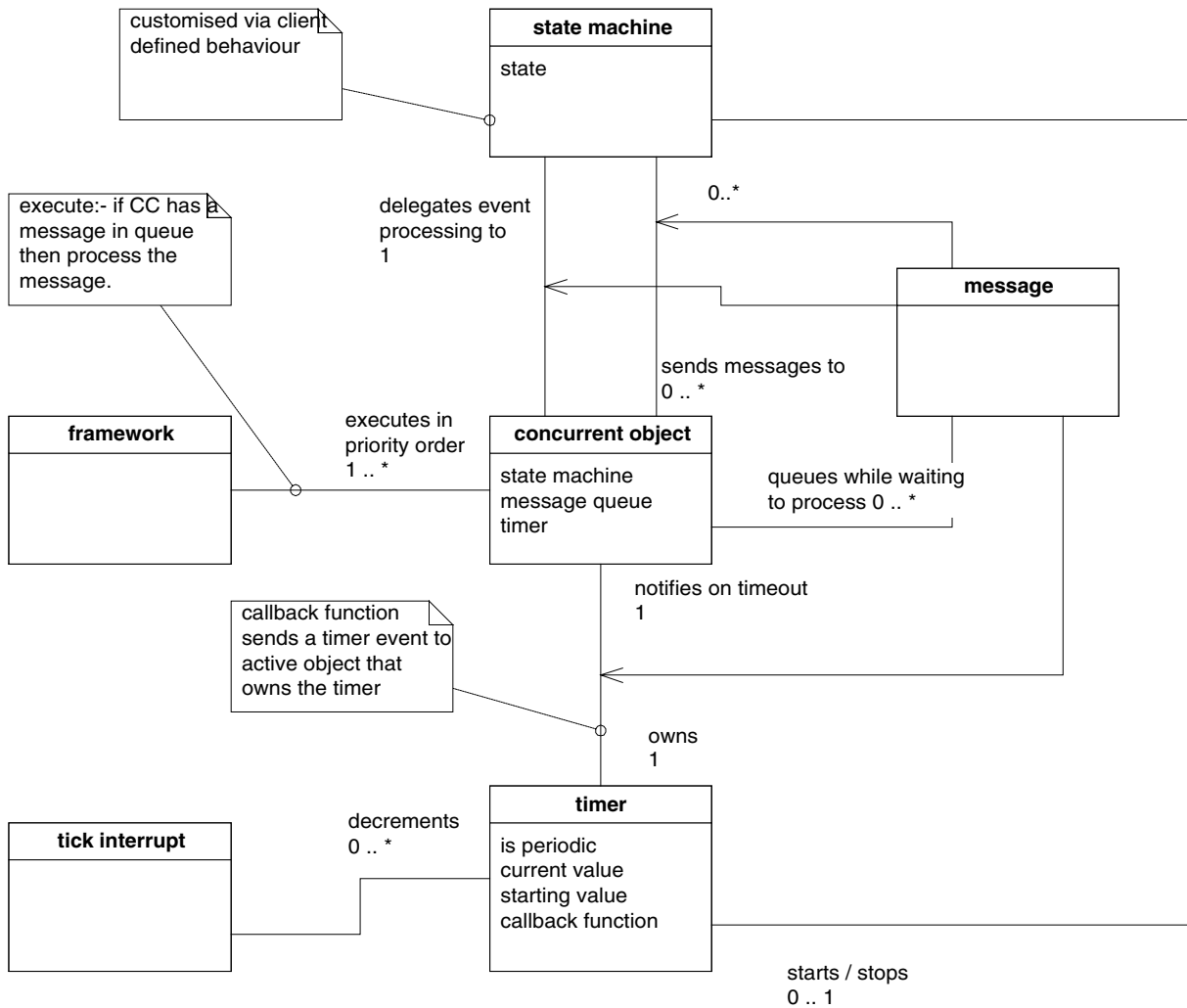
Advantages to user of code

- See how the framework is used by client software.
- Step through the code on desktop, and see how it works.
- Have some confidence in the level of probable defects - see below.

Advantages to code author

- Thinking of test cases - possible failure conditions - before writing the code leads to more robust code.
- The desktop is a more comfortable place to develop code
- A good test harness encourages the approach of "write a little, test a little, write some more..." Bugs are easier to find, because they are most likely to lie in the code most recently written.
- A good test harness encourages refactoring with confidence. Code will evolve over time, rather than devolve.

1.5 FRAMEWORK



A framework has a collection of concurrent objects. The framework executes concurrent objects in priority order. Execute means to respond to an event, if one is queued. The concurrent objects form a priority queue: objects at the head of the queue always execute in preference to objects further down the queue.

An event is a simple object whose primary attribute is its identity.

Concurrent objects have state behaviour. They respond to events by delegating event processing to a state machine. State machines are customised with client-defined behaviours: Customised state machines provide the application-specific state behaviours.

Amongst other things, concurrent object state machines may send events to other concurrent objects. This communication is asynchronous - the event is queued at the receiver concurrent object for later execution.

The state machine of an concurrent object may also start a timer. Like the state machine, the timer is owned by the concurrent object. In principle, a state machine could have several timers running - in practise, the framework provides other ways of getting multiple timers - so a maximum of one timer is supported by the concurrent object. Running timers are decremented by the tick interrupt.

Although a state machine requests that the timer be started, the operation is provided by the concurrent object. The timer can be one-shot, or periodic. The concurrent object supplies a call back function when it starts the timer: When the timer expires, it executes the call back function. The timer expiry function simply queues a timer event for the concurrent object.

1.6 EVENTS AND MESSAGES

- Needs to be a small type that is easy to copy.
- Lifecycle: Come into being at sender; consumed by receiver.
- Messages can carry data - eg. keyboard event, pointer to message data (communication system).
- User code must manage storage.
- A better framework would provide pools of fixed size buffers, and automatic allocation and deallocation.

Event is a simple scalar:

```
typedef uint8 event_t;
```

A message can be a simple event, or an event with a pointer:

```
#ifdef MESSAGE_QUEUES
```

```
typedef struct
{
    event_t id;                                /* event id */
    void * mptr;                               /* message data - clients allocate and delete */
} msg_t;
```

```
#else
```

```
typedef event_t msg_t;                       /* a message is a simple event */
```

```
#endif /* MESSAGE_QUEUES */
```

The identity of a message is an event:

```
#define id(msg) msg.id                       /* get (event) identity of msg */
#define id(e) e                             /* identity of event is event */
```

Create a message from an event:

```
msg_t msg (uint8 e);                         /* Turn an event into a message */
#define msg(e) e                             /* turn an event into msg and still have event */
```

1.7 MESSAGE QUEUE

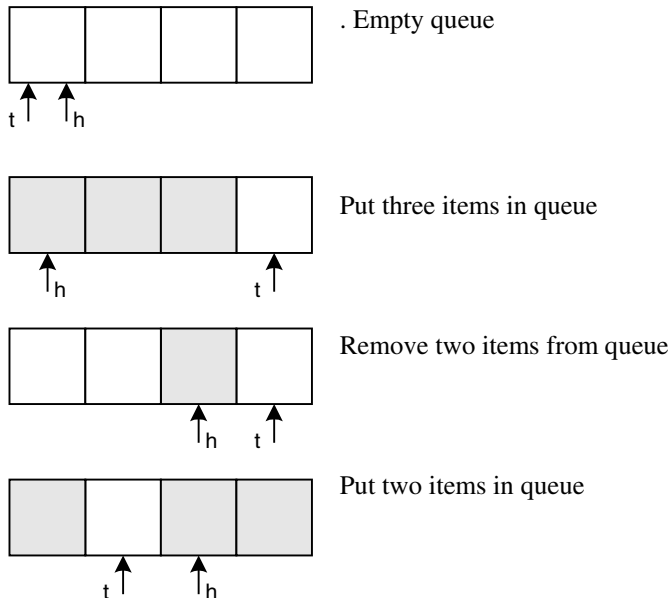
m_queue_t	
- head	: uint8
- tail	: uint8
- mask	: uint8
- p_buf	: msg_t []
+ mq_new (size) : m_queue_t *	
+ m_queue_delete ()	
+ mq_init (p_buf, size)	
+ mq_has_msg ()	
+ mq_put (msg)	
+ mq_get ()	: msg_t

.

- First-in, First-out. Circular buffer with head and tail indices.
- Buffer is a simple fixed size array.
- Head index points to the oldest data item in the queue. Items are removed from the head.

- Tail index points to the next empty position. New items are placed in the queue at the tail.
- If head and tail are the same, then the queue is empty.
- The head and tail indices wrap to zero after the end of the buffer.
- If the buffer is a power of 2 in size then wrapping the pointer can be achieved by AND operation with mask function:
mask = buffer size - 1 (eg: buffer size = 4; mask = 4-1 = 3).

```
p_buf [tail++] = m;
tail &= mask;      /* mask = buffer size - 1 */
```



1.8 ROLL YOUR OWN OBJECTS

Object Oriented systems have three characteristics:

- Encapsulation - what an object does. Objects bind together a set of attributes, and a set of behaviours. A class is a specification of an object.
- Inheritance - the ability to specify a class as an extension of an already existing class. A base class can define an interface shared by a set of classes.
- Polymorphism - objects can interact via interfaces defined by base classes. Objects do not need to know exactly which class their associate belongs to - the appropriate method is chosen at run time. (late / dynamic binding).

It is possible to do this stuff in C, but you have to roll your own:

A C struct is a collection of related data items:

```
typedef struct m_queue_t
{
    uint8    tail;          /* tail index */
    uint8    head;         /* head index */
    uint8    mask;         /* for overflow comparison */
    msg_t    * p_buf;      /* user allocated msg buffer */
} m_queue_t;
```

C methods are ordinary functions:

- The first parameter of a method call is a pointer to the object.
- Every reference to an instance variable is prefixed with the object pointer

C++:

```
void m_queue_t::mq_put (msg_t m)
{
    p_buf [tail++] = m;
    tail &= mask;           /* wrap index, if needed */
}
```

C:

```
void mq_put (m_queue_t * pmq, msg_t m)
{
    pmq->p_buf [pmq->tail++] = m;
    pmq->tail &= pmq->mask;   /* wrap index, if needed */
}
```

- These versions don't deal with queue full.
- The C technique is used internally by the C++ compiler. The two fragments above should produce identical assembly code.
- A good 'put' routine will deal with queue full:

```
void mq_put (m_queue_t * pmq, msg_t m)
{
    pmq->p_buf [pmq->tail++] = m;
    pmq->tail &= pmq->mask;   /* wrap index, if needed */

    /* If queue is full, lose oldest msg */
    if (pmq->tail == pmq->head) { pmq->head = ++pmq->head & pmq->mask; }
}
```

1.9 ALLOCATION AND INITIALISATION

- Permanent allocation is a good idiom for embedded systems - but should the framework or the client do the allocation and initialisation ?
- The client knows how many and what sizes are required - not the framework.
- Requiring that the client successfully allocate and initialise the framework is a serious breach of partitioning.
- This problem is the reason that dynamic storage allocation is so popular in object oriented languages - client requests that the framework allocate and initialise on demand.

Active framework provides two options:

- 1) Load time, by the client; framework makes available initialisation functions: Some messy construction details are now the responsibility of the client.
- 2) Run time, by the framework; Convenient for client - client calls new () function, which uses malloc () and then performs initialisation. new() function accepts size parameters as required.

```
/* Initialise a msg queue that is already created */
void mq_init (m_queue_t * pmq, msg_t * pbuf, uint8 q_size)
{
    assert (is_power_2 (q_size));

    pmq->tail = 0;
    pmq->head = 0;
    pmq->mask = q_size-1;
    pmq->p_buf = pbuf;
}

/* Create and initialise a msg queue (don't forget to call mq_delete()) */
m_queue_t * mq_new (uint8 size)
{
    event_t * pbuf;
    m_queue_t * pmq;

    pbuf = malloc ((size_t)size * sizeof (msg_t));
    assert (pbuf);
    pmq = malloc (sizeof (m_queue_t));
    assert (pmq);
    mq_init (pmq, pbuf, size);
    return (pmq);
}
```

1.10 DYNAMIC STORAGE AND EMBEDDED SYSTEMS

Disadvantages of malloc()

- Two new classes of fault: memory leaks and dangling pointers.
- Don't discover until run time that system may not have enough memory
- Larger heap than required wastes memory
- malloc () and free () take up code space
- malloc () and free () are generally not thread-safe (don't call malloc in an ISR !)
- malloc () and free () do not have deterministic execution time - malloc can take a long time to execute.
- Fragmentation may mean that malloc can fail, even when there is sufficient memory available.

Advantages of malloc()

- Its convenient

- It provides a solution to the encapsulation problem when using generic code.

How to use malloc() safely

- Allocate all storage at initialisation time, and don't release it.
- Use buffer pools for memory requests at run time: No fragmentation, constant alloc / dealloc time, thread safe. Code is simple and small.

1.11 RACE CONDITIONS

- Sending messages is a ubiquitous operation - it may be invoked by other threads, or interrupt service routines. The following line is from `mq_put()`

```
pmq->p_buf [pmq->tail++] = m;
```

The compiler will produce code that does:

- 1) Store msg `m` in the buffer at the tail index
- 2) Increment the tail pointer.

- Between (1) and (2), the queue is not consistent. An interrupt routine which puts a msg in the queue would cause chaos.
- This is a classic race condition.
- An easy solution is to prevent the critical region from being interrupted:

```
void mq_put (m_queue_t * pmq, msg_t m)
{
    disable_interrupts ();
    pmq->p_buf [pmq->tail++] = m;
    pmq->tail &= pmq->mask;          /* wrap index, if needed */

    /* If queue is full, lose oldest event */
    if (pmq->tail == pmq->head) { pmq->head = ++pmq->head & pmq->mask; }
    enable_interrupts ();
}
```

- Be careful when disabling interrupts - it can increase the worst case interrupt latency (add to the worst case system response time). Do not use this technique for protecting large blocks of code.
- Code which is safe in the presence of interrupts is called re-entrant. Any code which may be called from different contexts (thread, interrupt, non-interrupt) must be reentrant.
- Functions which only manipulate data on the stack are re-entrant: `strcpy()` is normally re-entrant. `strtok()` is not.
- Functions which manipulate global data will not be reentrant without some means of coordinating access.
- `mq_get()` is not re-entrant: It is called only by the concurrent object which owns the queue. It modifies the head index, but not the tail.

Rules

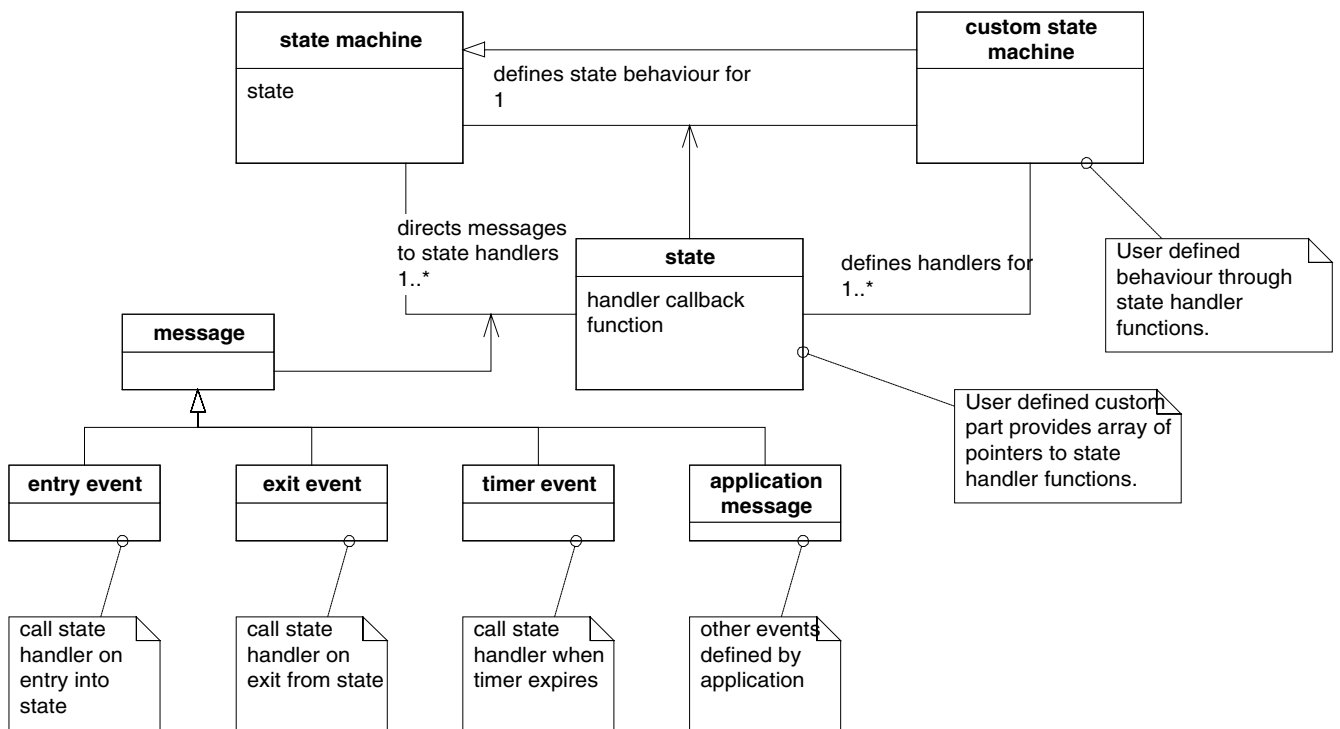
- Any routines which manipulate shared data need to have critical regions protected to make them reentrant.
- Disabling interrupts for long periods will affect the real time response of the system.
- Be careful about sharing permanent data between contexts - if necessary add tasks to manage shared resources.

1.12 STATE MACHINES

- State machines form a two-level decision hierarchy - action depends on current state, and received event.
- Goals for a state machine implementation:
 - Framework part provides the mechanism, and an interface into which to plug the actions.
 - Framework responsibilities: Initialisation, maintain the state, perform entry and exit actions, invoke the user defined actions.
 - Custom part provides application specific actions.

Common techniques:

- Nested switch statements - problems: Can get big monolithic code; client code must do all the work; no direct support for common actions (eg. entry and exit).
- State table - table of function pointers for each combination of state and event: Fast. Good separation between framework code (the code that traverses the tables), and client code (activated by the function pointers). Somewhat intricate for client to set up. Table can be large
- Individual states are sub classes of 'state' class (see Design Patterns - 'Gang of Four') - problems: Not extensible. Changing state requires destroying and creating an object.



Active framework provides a hybrid approach, borrowing from each of the techniques above.

An (abstract) state machine provides the interface to the rest of the framework, executing events on request. Maintains the state variable.

The custom state machine provides the application specific behaviours through a set of state handler functions - one for each state. The state machine calls the handler for the current state when it is requested to process an event. The state handlers are provided at initialisation time as an array of function pointers.

The custom state machine may extend the state machine by defining additional variables.

On entry to a state, the state machine sends an entry event to the state handler. The state handler should generate any I/O or other initialisation actions required for the state. On exit from the state, the state machine sends an exit event to the state handler. The state handler should perform any clean up required for the state.

1.13 (ABSTRACT) STATE MACHINE

Groundwork

- A state handler function takes a pointer to state machine, and a msg.

```
typedef void state_fn (sm_t *, msg_t);
```

- The identity of a message is an event enumeration.

```
enum { E_ENTRY, E_EXIT, E_TIME, E_APP };
```

.

sm_t	
- state	: unsigned
- state_table	state_fn []
+ sm_init (state_table, nr_states)	
+ sm_start ()	
+ sm_event (event)	
+ sm_msg (msg)	
+ sm_state ()	
# sm_goto (state)	

.

Receiving a Message

- Call the handler function for the current state, and pass it the message:

```
void sm_msg (sm_t * psm, msg_t m)
{
    psm->state_table [state] (psm, m);
}
```

Go to State (protected)

- Exit the current state, and enter the new one (implicitly recursive).

```
void sm_goto (sm_t * psm, state_t s)
{
    psm->state_table [state] (psm, msg(E_EXIT));
    psm->state = s;
    psm->state_table [state] (psm, msg(E_ENTRY));
}
```

Starting the state machine

- Enter the first state in the table

```
void sm_start (sm_t * psm)
{
    psm->state = 0;
    psm->state_table [state] (psm, msg(E_ENTRY));
}
```

1.14 DEFINING A CUSTOM STATE MACHINE

```
/* FSM has two states S1 and S2. Two events E1, E2. E1 stays in same state.
   E2 changes state. */
/*
   Creating a state machine:
   1) define the input events. Assign the first to E_APP
   2) Declare the state handler functions - one per state.
   3) Create and initialise a state table containing pointers to the handlers
   4) Declare an enumeration for the states that maps on to the state table.
   5) Declare any auxiliary variables that will be used by the state machine.
      The first (entry) handler function should initialise them.
   6) Define the handler functions - the entry event of the first state
      handler should perform initialisation
*/

enum { E1 = E_APP, E2 };

state_fn S1;
state_fn S2;

static state_defn_t States [] = { S1, S2 };
enum                { STATE1, STATE2 }; /* for sm_goto () */

void S1 (sm_t * psm, msg_t m)
{
    switch (id(m))
    {
        case E_ENTRY: /* do stuff */ break;
        case E_EXIT  : /* do stuff */ break;
        case E1      : /* do stuff */ break;
        case E2      : /* do stuff */ break;
    }
}

void S2 (sm_t * psm, msg_t m)
{
    switch (id(m))
    {
        :
        :
    }
}
```

1.15 EXTENDING THE SM: INHERITANCE IN C

- If custom SM defines its own variables, then it can create a struct which includes the (abstract) state machine - the custom state machine extends the abstract state machine.
- Can not rely on framework to allocate state machine - must do own allocation.
- To access own instance variables, the handler functions recast the state machine pointer - the state machine must be the first element of the struct.
- Only need to do this if *instances* of the custom state machine are required. If custom state machine is a singleton, then use simple file scope variables, and dispense with the pointer access.
- Singletons are natural idioms for small embedded systems, and C does singletons more naturally than C++.

1.16 FROM FSM TO HSM

- FSM = Finite State Machine - states do not nest.
- HSM = Hierarchical State Machine - states enclose other states - can be in several states simultaneously.
- The framework implements FSM or HSM selected by compile time flag DEF_HSM

Characteristics of HSMs

- 'Initial transitions' define internal transitions which are executed following state entry.
- Performing a state transition may involve exiting a number of states, and entering a number of states.
- If current state does not handle an event, enclosing state may do.
- Nesting information is provided by extra variables in state table: `enclosing_state`, and `level`

1.17 TIMERS

- Timers are useful in any real-time system; not just concurrent object ones.

timer list	timer_t
<pre>- Timer : timer_t *</pre>	<pre># remaining: uint32 # initial : uint32 # timer_fn # fn_arg # p_next : timer_t *</pre>
<pre>+ timer_tick () + timer_init () + timer_stop_all ()</pre>	<pre>+ timer_start (time, repeat, fn, arg) + timer_stop ()</pre>

- The timer list, `Timer`, is a singleton, which holds all of the running timers. The timer list functions directly access the timer variables.
- Timers on the list count down each time the `timer_tick ()` function is executed. `timer_tick ()` is normally called from a timer interrupt.
- `timer_start ()` places a timer on the list - it is the responsibility of the client to allocate and deallocate the storage for the timer.
- When `timer->remaining` counts down to zero, `timer_fn (fn_arg)` is executed. Beware - `timer_fn ()` may be called from an interrupt context.
- Useful things for `timer_fn ()` to do:
 - Active framework sends a timer event (`fn_arg` is a `sm_t *` in this case)
 - Set a flag for later checking (if using simple round - robin system).
 - Unlock a semaphore (if using a threaded RTOS) (Even if using an RTOS it is usually better to send an event).
- If `timer->initial` is non-zero, then the timer is restarted after it expires - get a repeating timer. Otherwise, timer is deleted from the list, and have a one-shot.
- All the timer routines disable interrupts to protect the integrity of the timer list while it is being modified. If user supplied handler functions take too long to execute, ticks can be lost.
- A better timer implementation could avoid losing ticks by enabling interrupts during timer processing, and protecting the timer list with a simple counting semaphore.
- This implementation optimises the per-tick processing: Only the timer at the front of the queue is decremented. Timers are added to the list in timeout order. Timers towards the back of the list are pre-decremented so that they expire after the correct number of ticks.

1.18 CONCURRENT OBJECTS

- Concurrent objects are held in a list, which is created and held by the application software.
- CC is an aggregate object, subsuming state machine, message queue and timer objects
- Objects execute when they have a message in their queue.
- The list holds the CCs in priority order: Objects at the front of the list execute in preference to the ones further back.
- The list functions and concurrent object functions are implemented in the same file - list access functions are friends of the concurrent object, and can access the internal variables.

concurrent object list
- list : concurrent_t *
+ cc_add_front (cc) + cc_start_all () + cc_run () + cc_delete_all ()

concurrent_t
sm : sm_t # mq : m_queue_t # timer : af_timer_t # p_next : concurrent_t *
+ cc_init (msg_buf, state_table []) + cc_new (eq_size, state_table []) + cc_delete () + cc_send_event (event) + cc_send_msg (msg) # cc_has_msg () # cc_of (:sm_t *) # cc_timer_start (time) # cc_timer_repeat (time) # cc_timer_stop ()

- Client or framework can allocate storage for the CC.
- CCs can start and stop timers if requested by the custom state machine.

1.19 INTERFACE TO THE RUN-TIME SYSTEM

The framework is designed to run within a thread on BrickOS - the Lego Mindstorms C OS. BrickOS requires:

- A well defined entry point.
- Connect a timer interrupt to the tick service routine: `timer_tick()`
- Give control back to the OS when there is nothing to do.
- Perform an orderly cleanup when asked to shut down.

Declarations: "af_impl.h, tick.h"

Entry Point

Setting up the entry point is the responsibility of the application. BrickOS tasks entry points are called `main ()`

Idle Processing

```
typedef unsigned long wakeup_t; /* generic parameter for wakeup function */
typedef wakeup_t wakeup_fn (wakeup_t); /* Wake up function */
```

```
wakeup_t wait_event (wakeup_fn * pwf, wakeup_t data);
```

- When there are no events, the framework has nothing to do. It calls `wait_event()` - the 'idle' function.
- `wait_event()` accepts a pointer to a call back function: `pwf()` returns true when a concurrent object has an event waiting.
- Run time system must call wake up function `pwf()` regularly. If `pwf()` returns true, then `wait_event()` is complete.

```
wakeup_t wait_event (wakeup_fn * pwf, wakeup_t data)
{
    wakeup_t temp = 0;
    while (!temp) { temp = pwf (data); }
    return (temp);
}
```

Orderly Exit

- BrickOS provides a function `shutdown_requested ()` which returns true when the application should exit. The framework calls `shutdown_requested ()` after processing each event. If `shutdown_requested ()` evaluates true, the main loop of the framework terminates.
- The run time system must implement `shutdown_requested ()`

Framework main loop

```
/*
    Find cc objects with msgs and execute them in priority order.
    if no msgs available, then call wait_event ()
    if shutdown_requested () then exit
    pcc is the head of list of concurrent objs
*/
void cc_run (concurrent_t * pcc)
{
    concurrent_t * p_ready;          /* the one returned by wait_event() */

    for (;;)          /* find the highest priority cc with msg and execute it */
    {
        while ((p_ready = msg_avail (pcc)) != 0)
        {
            if (shutdown_requested ()) { return; }
            cc_exec (p_ready);
        }

        p_ready = (concurrent_t *)          /* find a ready cc obj */
            wait_event (msg_avail, (wakeup_t)pcc);

        if (shutdown_requested ()) { return; }
        if (p_ready) { cc_exec (p_ready); }
    }
}
```

1.20 DINING PHILOSOPHERS

The Shengdu monastery in Tibet is a place for contemplation. The philosophers there ponder the world's great questions. Thinking takes a lot of energy, so they sit around a table, where food is always supplied. But too much food makes philosophers sleepy, not contemplative, so the monastery management supplies only as many forks as philosophers - they need two to eat:

- Philosophers eat until they are no longer hungry, at which time they begin thinking.
- After thinking for some time, they become hungry, and wish to eat. They can't think while they are hungry.
- Philosophers need two forks to eat - they have one on each side - but there are only as many forks as philosophers. If a philosopher is eating, the philosophers on either side can not eat.

Write a simulation using the active framework:

- If two philosophers are able to acquire the same fork, the simulation has failed.
- If all of the philosophers end up waiting for one fork, the simulation has failed.
- The simulation should minimise the amount of time that a philosopher is hungry.

Advice:

Simulate the passage of time by creating a tick object at the lowest priority (last in queue). It repeatedly calls the tick entry function, and then sends a message to self.

Write a function to display the application states, and call it whenever the state changes.