A Single Threaded Software Architecture for Embedded Systems

G. Andrew Mangogna

May 9, 2017
Version 1.8

Abstract

This paper is a literate program for a set of software architecture mechanisms that constitute the run time support for a single threaded software architecture domain which is suitable for a large class of applications found in embedded computer systems. The document discusses and explains all the code found in the mechanisms and, as a literate program, can be transformed into the source code for the mechanisms.

Contents

List of Figures

1 Introduction

The *Software Architecture Domain* has always been special in Executable UML. It is the domain that specifies the policies for managing data and execution for the system and is the target of model translation. Much of the work of model translation is to generate code in the target implementation language that weaves together the logic of the model actions with invocations of domain operations of the Software Architecture Domain to accomplish the model semantics.

This paper is a literate program for a simple Single Threaded Software Architecture. It implements what can be considered a virtual machine that obeys the execution rules for Executable UML. The implementation language is "C". The mission of a software architecture domain is to implement a set of policy decisions about how data is managed and how execution is sequenced for any system built upon this architecture. The policy decisions made here are oriented toward highly embedded systems that are typically deployed on very minimal computing platforms. As we shall see, the architecture can also be run on POSIX platforms with the primary goal being simulation of the embedded platform.

Policy decisions intended for an embedded micro-controller based platform are unlikely to be optimal for more conventional computing platforms. So we make no claims of universality for this architecture. Indeed it is easy to envision applications which would not be suitable to deploy on a single threaded architecture where there are no separately scheduled execution contexts. It should come as no surprise then that there is not a single Software Architecture Domain suitable for all classes of applications. That is not the point here. What we are trying to accomplish here is to show the implementation of a set of mechanisms for a Software Architecture Domain that does not depend upon the specifics of an application's functionality and is appropriate for any application whose computational demands reasonably match the design criteria of mechanisms.

To divorce the management of data and execution from the application functionality is a more radical concept than might be first recognized. In this paper will we discuss many aspects of execution without any specific reference to what a particular application is attempting to accomplish from a functional point of view. This tends to make the discussion rather abstract and can make it harder to envision how this code might be put to use in any given software application. But remember, this code represents a run-time engine that is the final target of translating an XUML model into a running system. We are discussing a well encapsulated world of data and execution management that is, however unfortunate, completely devoid of any application semantics. This notion often makes many programmers uncomfortable in their belief that everything about a software system is to be tailored to the specific semantics of the application functionality.

The design of STSA is guided by the following criteria:

- There is a very limited budget for running the computing hardware. Often this limited budget arises from limitation of supplying power to the computing hardware or the need to minimize the costs of the hardware. Consequently, the software execution framework must closely match the facilities directly provided by the hardware. Otherwise, providing capabilities that are not well supported by the hardware platform implies additional software execution that cannot be supported by the computation budget.

- Memory is very limited and typically segmented between that which is read only and that which is read/write with the later type of memory being particularly in short supply.

Certainly, modern computing environments do not tend to suffer from either of these limitations. So we are purposefully targeting limited computing environments as are found in highly embedded control systems and battery powered systems. Often these systems are of a safety critical nature where having very precise control over the software is essential.

1.1 Scope

This literate program discusses many concepts related the Executable UML. There is not space here to give you all the background of this methodology. The reader is referred to some of the many books available on Executable UML for the necessary background [1] [2] [5] [3] [4].

2 Main Loop

Let's jump right to the part that you are probably most interested in, the *main loop*. Figure 1 (p. 6) shows a flow diagram of the main loop. The program is an infinite loop. The mechanisms supply the required "C" program function, `main`. As discussed in the introduction, all the decisions about sequencing program execution are factored away from the application code and incorporated into the mechanisms. After initialization, we enter the loop. The first step is to synchronize to any asynchronous processing that has gone on. Asynchronous processing is discussed in detail in section 6 below (p. 94), but for now we can think of it as interrupts going off and needing to tell the rest of the program that something has happened. The way asynchronous execution does this is to post a function to the sync queue. This constitutes a request that the function be invoked at the first *safe* opportunity during the background processing. From the flow diagram, we see that all sync functions are removed from the sync queue before we consider dispatching any events to a state machine. Although there are no restrictions on what a sync function may do, typically it executes some domain operation that results in generating events to some class instance. Those generated events are placed on the event queue.

After the sync queue is empty, the loop then removes one event from the event queue and dispatches it. In general, dispatching an event will cause a state machine to transition and some state action will execute. State actions may cause other events to be generated. It is required that state actions leave the domain data in a consistent state. We can now see, that outside of a state action is a *safe* time to execute the synchronization functions.

Eventually, the thread of control initiated by some sync function will die out of its own accord. Some state actions perform computations without generating new events and eventually all the events are consumed. At that time, the sync queue will be empty and the event queue will be empty. There is nothing more to do and we then wait. Exactly what it means to *wait* is platform dependent as we shall see.

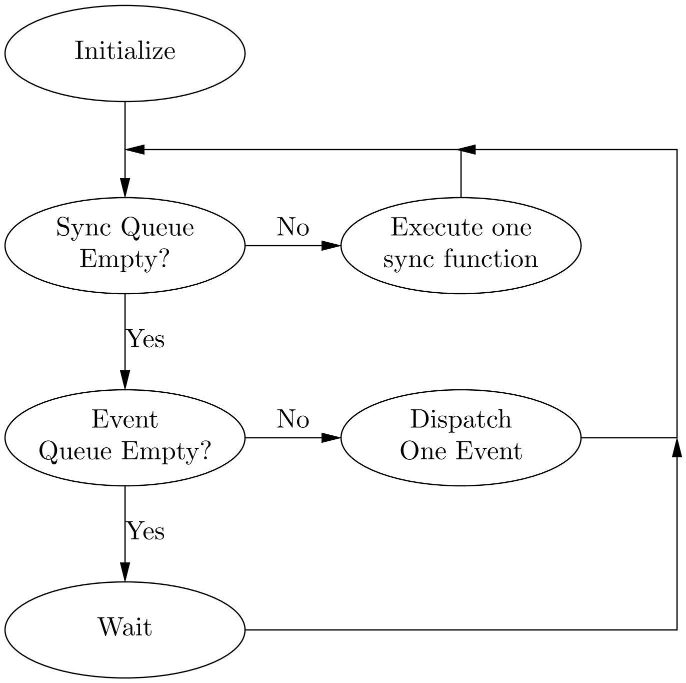

Figure 1: Main Loop Flow Diagram

We now examine how the general ideas of the main program flow actually get implemented in code. By default, he mechanisms define the program entry point given by the "C" function `main`. For testing it is convenient to allow the `main` definition to be elsewhere and we achieve that via conditional compilation.

6 ⟨*external test functions* 6⟩≡ (149 152 155) 100b ▷

```
extern void stsa_main(void) ;
```

7 ⟨*main loop* 7⟩≡ (150 153 156)

```
#ifdef MECH_TEST
void
stsa_main(void)
#else
int
main(void)
#endif /* MECH_TEST */
{
    mechInit() ;

    for (;;) { /* Infinite Big Loop */
        /*
         * Empty the foreground/background sync queue.
         */
        ⟨background sync 8⟩
        /*
         * Dispatch one event from the event queue.
         */
        if (!mechDispatchOneEvent()) {
            /*
             * Check if this thread of control is complete
             * and wait if there is no additional work to
             * be done.
             */
            mechWait() ;
        }
    }
}
```

Not surprising, the `main` function is an infinite loop. After initialization, the infinite loop is entered. First, we must perform any synchronization between asynchronous foreground execution and the single background thread. This is described in detail in section 6 (p. 94). In most circumstances that means executing the synchronization functions that the foreground processing has requested. The synchronization functions are run as part of the background processing. Also note that the sync queue is emptied before considering any events. This design gives foreground / background synchronization a higher effective priority than the thread of control that might be generated by executing a sync function. For most systems that synchronization is given a simple loop.

8 ⟨*background sync* 8⟩≡ (7)

```
#ifndef __ARM_ARCH_7M__
/*
 * Empty the foreground / background
 * synchronization queue.
 */
while (mechInvokeOneSyncFunc()) {
    ; /* empty */
}
#endif /* __ARM_ARCH_7M__ */
```

As we shall see below (section 10.3, p. 128), for the Cortex-M3 this synchronization is not executed in the main loop but rather in an exception handler. More on this particular processor architecure later, but it does not change the essential characteristic that synchronization occurs first before events are dispatched.

After synchronization is complete, one event is dispatched. The actions executed upon the state transition caused when the event is delivered may very well generate other events into the event queue. However, after each dispatched event, we determine if there is any synchronization that must occur. In this way, events are removed and dispatched from the event queue one at a time checking if there are any sync functions that must be executing in between. Note that in this design, the event queue is not placed in a critical section. Interrupts are *not* allowed direct access to the event queue. Asynchronous execution has only one method to request computation and that is the sync queue.

Eventually, we will run out of work to do[1]. It is then time to wait for the next thing to come along. What it means to wait is also platform dependent. For bare metal systems, particularly low power ones, the processor will be put to sleep to save power. On an operating system, there are system specific means to cause the process to sleep. The implementation must be careful in determining that we indeed are ready to wait. We must be able to test for an empty sync queue and wait as a single atomic action. We will see how that happens in section 9.5 (p. 118) below.

An important thing to remember here is that this represents all the flow of execution policy for the entire system. No part of the application code determines when a state machine event is dispatched. Too many times, code which decides what to do next is scattered throughout a program. This code is often inconsistent, many times just wrong and carries a large testing and maintainence burden. It is unnecessary to do that. Careful factoring of execution sequencing away from the application code can place all the policy decisions in a very small amount of code. Your system may need different a policy to meet the computational demands of your applications, but that does not mean that the execution policy must be tailored for every distinct application. Software architectures can meet the computation requirements of large classes of applications and do so in a manner that is independent of the subject matter of the application.

The careful reader will have already recognized that there is no way to exit the program directly. This is in keeping with the highly embedded nature of the class of applications for which STSA is intended. Your system may need a way to terminate and if it does, then you should insert the termination check just before the call to `mechWait`. This allows the ongoing thread of control to complete before termination.

[1]or else we have some type of unbounded computation

3 Managing Data

The STSA was designed for highly embedded systems. These types of systems often tend to be **long running**, *i.e.* the program is intended to run for the lifetime of the system without ever stopping. This means that there is no opportunity to restart and clean up if something goes wrong. Safety critical systems, for example, just do not have that luxury. The long running nature of embedded applications has led to the general notion among embedded programmers that a central system heap is a bad idea. A heap can fragment and at critical times not have sufficient contiguous memory to allocate an essential data structure. Whether or not use of a heap is a bad idea in these types of systems is subject to some skepticism. I don't know of any definitive studies, but the conventional wisdom is well entrenched.

What is used instead is fixed, worst case allocation. It is less efficient in the way that memory is utilized, but tends to be more deterministic in the way that it behaves. A heap can introduce probabilistic behavior. At least with fixed allocation and a good set of test cases, you can have reasonable assurances that memory resources are properly allocated and released.

As you will see below, all data is allocated in fixed sized pools, the size of which are determined at compile time. This includes internal data structures as well as those holding the application instance data. This may take some of the non-deterministic behavior out of the allocation, but it does not make the job of estimating the proper sizes of the memory pools any easier. This is usually accomplished by simple characterization, *i.e.* testing it to failure and determining the best size for things..

3.1 Class Instances

In order to keep track of the essential status of instances, all instance structures include a data structure as its first member. This is the view that the mechanisms take of an instance and the mechanisms cast all instance pointers to this data type. In object oriented parlance, you can think of this as the base class for all instances.

10 ⟨*data types* 10⟩≡ (149 152 155) 12 ▷
```
typedef struct mechinstance {
    AllocCount alloc ;
    StateCode currentState ;
    struct mechclass const *instClass ;
} *MechInstance ;
```
Defines:
`MechInstance`, used in chunks 12, 14–16, 18b, 19, 24, 31–34, 37, 38, 44–49, 65, 75, 78, and 84–86.

The `alloc` member is used to determine if a particular storage slot is currently in use. If `alloc` is 0, then the slot is free. A non-zero value indicates that an instance is allocated to that slot. This member also plays an important role in detecting the *event-in-flight* error as we shall see below (see section 4.4.1, p. 68). A small 8-bit quantity is sufficient for this member.

11a ⟨*base types* 11a⟩≡ (149 152 155) 11b ▷

```
typedef uint8_t AllocCount ;
```

Defines:
`uint8_t`, used in chunk 139.

The `currentState` member holds the value of the current state for the instance.

11b ⟨*base types* 11a⟩+≡ (149 152 155) ◁ 11a 21 ▷

```
typedef uint8_t StateCode ;
```

Defines:
`uint8_t`, used in chunk 139.

As we see from the data type, states are held in 8 bit quantities and this limits the number of states that a class may have to 253. Why 253? Two of the values in the state type are used to indicate if a given event is to be ignored or treated as a fatal error.

11c ⟨*constants* 11c⟩≡ (149 152 155) 22 ▷

```
#define MECH_STATECODE_IG UINT8_MAX
#define MECH_STATECODE_CH (UINT8_MAX - 1)
```

Defines:
`MECH_STATECODE_CH`, used in chunk 65.
`MECH_STATECODE_IG`, used in chunk 65.

We use the two highest values of a `StateCode` to indicate ignored and can't happen, respectively. The semantics of ignoring an event and deeming an event as not possible to happen are explained below. In practice, 253 states would be an enormous state model and clear indication that you should rethink your design.

The `instClass` member is a pointer to a data structure that defines the class. The class data structure is discussed below (section 4.4.4, p. 79). For now, we can think of it as the data structure that holds all the information that is invariant for the class, *i.e.* the information that applies to all instances of the class.

3.2 Instance Allocation

There are three ways to create an instance:

1. Create an instance as part of an initial instance population.
2. Create an instance synchronously to the execution of some action by invoking a function.
3. Create an instance asynchronously to the execution of some action by sending an event.

The initial instance population consists of defining the storage pool for the class to have initial values. In "C" this is nothing more than defining an array with initializers, since we use simple "C" arrays as the storage pools. This can be a tedious undertaking, but the mechanisms provide no support for these definitions since this is really a compile time undertaking. Tools such as `pycca`[2] can significantly ease the tedium. In this section we are going to discuss synchronous instance creation. This is instance creation by a direct function invocation. Later we will discuss asynchronous instance creation which is instance creation by generating an event.

Instance allocation also supports a very simplified notion of construction and destruction for the instances. This is no where near as complicated or full featured as something in C++. Constructors and destructors take no arguments other than a pointer to the instance.

12 ⟨*data types* 10⟩+≡ (149 152 155) ◁10 13▷
```
  typedef void (*InstCtor)(MechInstance) ;
  typedef void (*InstDtor)(MechInstance) ;
```
Defines:
`InstCtor`, used in chunk 13.
`InstDtor`, used in chunk 13.
Uses `MechInstance` 10.

[2]See http://tcl-cm3.sourceforge.net/

It is primarily useful for when the instance has a more complicated data structure as an attribute. If you need to do complicated construction of instances, the preferred method is to do that with an instance based operation or as part of a state action for an asynchronously created instance.

To support managing a pool of class instances, an **Instance Allocation Block**, or **IAB** for short, data structure is used to keep track of the memory pool.

13 ⟨*data types* 10⟩+≡ (149 152 155) ◁12 23a▷

```
typedef struct installocblock {
    void *storageStart ;
    void *storageFinish ;
    void *storageLast ;
    AllocCount allocCounter ;
    size_t instanceSize ;
    InstCtor construct ;
    InstDtor destruct ;
} *InstAllocBlock ;
```

Defines:
`InstAllocBlock`, used in chunks 14, 16–19, and 79b.
Uses `InstCtor` 12 and `InstDtor` 12.

What we need is a data structure that can find the bounds of the pool.

`storageStart` A pointer to the beginning of the memory where the instance storage pool is located. Typically this is an array allocated to hold the instances of a class.

`storageFinish` A pointer to one element beyond the end of the instance storage pool for the class. This pointer may not be dereferenced, of course, but provides the boundary marker for the end of the pool.

`storageLast` A pointer to the instance that was last allocated. This is used as the starting point for allocating the next instance.

`instanceSize` The number of bytes of memory occupied by an instance.

`construct` A pointer to a constructor function. If there is no constructor defined for the class, then the value of this member may be set to `NULL`.

`destruct` A pointer to a destructor function. If there is no destructor defined for the class, then the value of this member may be set to `NULL`.

As we shall see, the underlying allocation algorithm is a simple linear search.

14 ⟨*instance functions* 14⟩≡ (150 153 156) 16 ▷

```
static MechInstance
mechInstFindSlot(
    MechClass instClass)
{
    assert(instClass != NULL) ;
    InstAllocBlock iab = instClass->iab ;
    if (iab == NULL) {
        return NULL ;
    }
    assert(iab->storageLast < iab->storageFinish) ;
    /*
     * Search for an empty slot in the pool.
     */
    MechInstance inst ;
    for (inst = mechInstNext(iab, iab->storageLast) ;
            inst->alloc != 0 && inst != iab->storageLast ;
            inst = mechInstNext(iab, inst)) {
        /*
         * Empty
         */
    }
    /*
     * Check if we ended up on a slot that is not allocated.
     */
    return inst->alloc == 0 ? inst : NULL ;
}
```

Uses `InstAllocBlock` 13 and `MechInstance` 10.

3.2.1 mechInstCreate

The function, `mechInstCreate`, is used to synchronously create an instance of a class.

15 ⟨*external scoped functions* 15⟩≡ (149 152 155) 18b ▷

```
extern MechInstance mechInstCreate(
    MechClass instClass,
    StateCode initialState) ;
```

Uses `MechInstance` 10.

It takes as arguments:

`instClass` A pointer to the class structure for the instance to be created.

`initialState` The state number into which the instance will be placed. For classes that do not have an associated state model this argument should be set to 0.

The return value of the function is a pointer to the new created instance. Since the return type of the function is `MechInstance`, in general it will have to be cast to be a pointer to the correct class structure type.

16 ⟨*instance functions* 14⟩+≡ (150 153 156) ◁14 19▷

```
MechInstance
mechInstCreate(
    MechClass instClass,
    StateCode initialState)
{
    assert(instClass != NULL) ;
    InstAllocBlock iab = instClass->iab ;
    #ifndef NDEBUG
    if (instClass->odb) {
        assert(initialState < instClass->odb->stateCount) ;
    }
    #endif /* NDEBUG */

    /*
     * Search for an empty slot in the pool.
     */
    MechInstance inst = mechInstFindSlot(instClass) ;
    if (inst == NULL) {
        mechFatalError(mechNoInstSlot, instClass) ;
    }
    /*
     * Record where we left off for the next time around.
     */
    iab->storageLast = inst ;
    /*
     * Mark the slot as in use.
     */
    inst->alloc = mechInstIncrCounter(iab) ;
    inst->currentState = initialState ;
    inst->instClass = instClass ;
    /*
     * Run the constructor if there is one.
     */
    if (iab->construct) {
        iab->construct(inst) ;
        }
        return inst ;
    }
```

Uses `InstAllocBlock` 13 and `MechInstance` 10.

As we can see it is a simple linear search of the pool looking for an empty slot to allocate. The `storageLast` member of the IAB is used as a starting location for the search. This attempts to improve the search speed on the assumption that the next free slot is most likely to be in the location after the last allocated one. It is a fatal system error to run out of instance storage space.

Since the pool is allocated in a contiguous block of memory, we must wrap around the iterator when it passes the end of the storage pool. That is accomplished with the `mechInstNext` function.

17 ⟨*instance allocation helper* 17⟩≡ (150 153 156) 18a ▷

```
    static inline
    void *
    mechInstNext(
        InstAllocBlock iab,
        void *ptr)
    {
        ptr = (char *)ptr + iab->instanceSize ;
        if (ptr >= iab->storageFinish) {
            ptr = iab->storageStart ;
        }
        return ptr ;
    }
```

Defines:
  `inline`, used in chunk 101b.
Uses `InstAllocBlock` 13.

It is very important to understand that a synchronously allocated instance does *not* run a state action. The instance may be placed into any valid state for the class, but as is seen above, no code is run other than the constructor. This is by design. If you need state action code executed when an instance is created then it is necessary to use asynchronous instance creation by generating a creation event. Alternatively, an event can be generated to the newly created instance immediately after it has been constructed. Both mechanisms have their uses, although the synchronous instance creation is probably overused in many designs since it is what most programmers are familiar with.

One other important point here. There is a counter in the IAB that is incremented each time an instance is allocated and this value is used in the `alloc` member of the instance. This is another part of the strategy to detect an *event-in-flight* error. This is described further below (p. 68). The effect of running this counter is that every instance gets a different `alloc` member value (modulo 255, naturally). The increment has one little catch. You cannot increment the counter and leave it at zero.

18a ⟨*instance allocation helper* 17⟩+≡ (150 153 156) ◁17

```
static inline
AllocCount
mechInstIncrCounter(
    InstAllocBlock iab)
{
    /*
     * Catch any wrap around to zero.
     */
    if (++iab->allocCounter == 0) {
        ++iab->allocCounter ;
    }
    return iab->allocCounter ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `InstAllocBlock` 13.

3.2.2 mechInstDestroy

The function, `mechInstDestroy`, is used to synchronously destroy an instance.

18b ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁15 31b▷

```
extern void mechInstDestroy(
    MechInstance inst) ;
```

Uses `MechInstance` 10.

It takes a single argument:

`inst` A pointer to an instance to be destroyed.

Just as there was a distinction between synchronous and asynchronous instance creation, there is a similar distinction for destruction. Asynchronous destruction happens as a result of an instance entering an **final** state and that is discussed further below. Here we are dealing with synchronous destruction of an instance.

19 ⟨*instance functions* 14⟩+≡ (150 153 156) ◁16 94▷

```
void
mechInstDestroy(
    MechInstance inst)
{
    InstAllocBlock iab = inst->instClass->iab ;
    assert(iab != NULL) ;
    /*
     * Run the destructor, if there is one.
     */
    if (iab->destruct) {
        iab->destruct(inst) ;
    }
    /*
     * Mark the slot as free.
     */
    inst->alloc = 0 ;
}
```

Uses `InstAllocBlock` 13 and `MechInstance` 10.

Destroying an instance is a very simple matter indeed. If there is a destructor, it is run. The slot is free when its `alloc` member has a value of 0. But beware, for designs that have complicated relationships among the classes, instance deletion can be very complicated, requiring much care that the interdependencies among classes are properly preserved. That work is not done here! You will have to include it in your action code.

4 Managing Execution

In this section we discuss the rules and policies associated with managing the sequencing of execution. There are two means available to an application to control the sequencing of execution.

1. Invoke an ordinary function.
2. Generate an event to the instance of a class.

Not much needs to be said about invoking functions. Computation transfers to the entry point and runs until the function is complete, transferring control back to next statement of the caller. Typically, such functions are organized into those that are associated with the domain as whole, a particular class or instances of a class. Such organization may be helpful to the programmer, but since they are directly supported by the implementation language, STSA does not get involved in mediating them.

Where STSA does get involved is for those computation that must leave off at some point, waiting for some other action in the system or the external environment, and then resume execution maintaining the past history. This type of execution is implemented as a state machine.

4.1 State Machine Rules

Each class that has lifecycle behavior may have a state model associated with it and each instance of that class will have a state variable. STSA supports a Moore type state model.

In the Moore formulation of state models, action code is associated with states and is executed upon entry into a state. This is distinguished from the Mealy formulation where actions are associated with the transitions and are executed upon exiting a state. Much writing and discussion has been wasted attempting to justify one type of state model over another. What we know is that they are computationally equivalent, *i.e.* we can prove that there is no problem that you can solve with a Moore machine that cannot also be solved with a Mealy machine and *vice versa*. Whether your application is easier to describe with one type rather than the other is something that you alone may decide. Moore machines are the traditional formulation for Executable UML and they have the simplest implementation structures. What we specifically reject here is any use of hierarchical state models. They are unnecessary and add complication that is not welcome. The power of computation in Executable UML is derived from the interaction of simple state machines each of which is tied to the lifecycle of a particular class. If you have some state model that is large and complicated where you think some other kind of higher order structure is needed, the usual reason is that you have multiple classes masquerading as one and further refinement of your analysis is necessary.

Generally, state actions affect other computations in the domain by updating instance attribute values or by generating events to other instances. The important distinction here is that the application code of the state actions does not deal with actually dispatching the events nor does it control which event is dispatched next.

4.2 Event Generation

There are three types of events:

1. Ordinary events that cause transitions in state machines.
2. Polymorphic events that are mapped at runtime across a generalization hierarchy.
3. Creations events that support asynchronous instance creation.

We will have need to distinguish the various event types and define and enumeration to do that.

21 ⟨*base types* 11a⟩+≡ (149 152 155) ◁11b 23b▷

```
typedef enum {
    NormalEvent,
    PolymorphicEvent,
    CreationEvent
} MechEventType ;
```

Defines:
`MechEventType`, used in chunks 24, 31a, 74a, 81, and 85.

The process of generating an event involves the following steps:

1. Obtain an **Event Control Block** from the free pool of ECB's.
2. Set the values of the fields in the ECB.
3. Queue the ECB for later dispatch.

4.2.1 Event Parameter Storage

Before we can talk about what an ECB contains, we need to deal first with events that carry parametric data. In this formulation of state machines, events may carry additional parameters. Clearly space has to be allocated for that data. The more difficult issue is to deal with the type of the parameter data. There are a couple of solutions, neither of which is very satisfying. We could collect all the parameters from all the state machines in the system and create a giant union. This would properly allocate the amount of parameter storage required and provide a type safe manner to deal with that data. Unfortunately, the parameters to states are scattered in very many places in a system and gathering them together is a difficult undertaking.

Here we take the view of providing some generic parameter fields with a fixed amount of memory and letting state actions cast that memory into the appropriate type. Needless to say, this can also be a source of errors, but is much easier to manage. Choose the technique that makes the most sense for your system. In many systems, the number of states that use parametric data is small and using a fixed size works better than might be expected. The important point here is that events can carry data with them. Many state machine formulations don't support this and it is very difficult to correctly manage memory lifetime without it. It is one of those things that you might not use very often but it is difficult to do without when you need it.

We fix the amount of memory used for event parameter storage, allowing it to be overridden by the compiler command.

22 ⟨*constants* 11c⟩+≡ (149 152 155) ◁11c 28a▷

```
#ifndef MECH_ECB_PARAM_SIZE
#   define MECH_ECB_PARAM_SIZE  16
#endif /* MECH_ECB_PARAM_SIZE */
```

Here we define a union of arrays for some common types. It is not exhaustive (*N.B.* no floats or doubles).

23a ⟨*data types* 10⟩+≡ (149 152 155) ◁13 63▷

```
typedef union {
    signed char cparm[MECH_ECB_PARAM_SIZE] ;
    unsigned char ucparm[MECH_ECB_PARAM_SIZE] ;
    unsigned short usparm[MECH_ECB_PARAM_SIZE / sizeof(unsigned short)] ;
    signed short isparm[MECH_ECB_PARAM_SIZE / sizeof(signed short)] ;
    unsigned uparm[MECH_ECB_PARAM_SIZE / sizeof(unsigned)] ;
    int iparm[MECH_ECB_PARAM_SIZE / sizeof(unsigned)] ;
    unsigned long ulparm[MECH_ECB_PARAM_SIZE / sizeof(unsigned long)] ;
    signed long ilparm[MECH_ECB_PARAM_SIZE / sizeof(signed long)] ;
    void *pparm[MECH_ECB_PARAM_SIZE / sizeof(void *)] ;
} EventParamType ;
```

Defines:
`EventParamType`, used in chunks 24 and 96c.

State actions may use the fields in the above union, or cast the whole thing to a type they like better. It is necessary that the code that generates the event and the code that consumes the event have the same idea of how the data will be treated.

4.2.2 Event Control Block

With that diversion done, on to the Event Control Block itself. The ECB is the primary data structure for generating and dispatching events.

First some data types. Events are reference counted and 8 bits are deemed sufficient for that. More on reference counting ECB's below (p. 25).

23b ⟨*base types* 11a⟩+≡ (149 152 155) ◁21 23c▷

```
typedef uint8_t RefCount ;
```

Defines:
`uint8_t`, used in chunk 139.

Events are also encoded as small zero based sequential integers.

23c ⟨*base types* 11a⟩+≡ (149 152 155) ◁23b 40▷

```
typedef uint8_t EventCode ;
```

Defines:
`uint8_t`, used in chunk 139.

Event queuing is done by doubly linked lists and the links are allocated as part of the ECB as the `next` and `prev` members. We also allocate a separate ECB structure to use as the event queue terminus.

24 ⟨*event data types* 24⟩≡ (149 152 155)

```
typedef struct mechecb {
    struct mechecb *next ;
    struct mechecb *prev ;
    RefCount referenceCount ;
    EventCode eventNumber ;
    AllocCount alloc ;
    MechEventType eventType ;
    union {
        MechInstance targetInst ;
        MechClass targetClass ;
    } instOrClass ;
    MechInstance srcInst ;
    MechDelayTime delay ;
    EventParamType eventParameters ;
} *MechEcb ;
```

Defines:
`MechEcb`, used in chunks 26, 27, 29–36, 41, 42, 46, 48–50, 52–56, 59, 61, 62, 65, 75, 78, 102c, and 130b.

Uses `EventParamType` 23a, `MechDelayTime` 40, `MechEventType` 21, and `MechInstance` 10.

An ECB is reference counted and that count is stored in the `referenceCount` member. Normally, all the reference counting is handled internally by the event generation and dispatch code. However, it is sometimes the case that a given event will need to be generated repeatedly. Rather than go through all the steps of event generation, you can use the reference count to prevent an ECB from being returned to the pool. By allocating the ECB, either from the pool using `mechEventAlloc` or as an ordinary variable, and then incrementing the reference count, the mechanisms will not return the ECB to the free pool after the event is dispatched. This also allows the ECB to be allocated in memory that is not in the normal ECB pool that is maintained by the mechanisms. Note that this is an optimization that carries with it the potential for bleeding ECB data structures and loosing them forever along with a number of other abuses of variable lifetime. It should be used with care and certainly if you need to generate high rate periodic events you should consider different design options (such as dedicated hardware timers).

The `eventNumber` member holds the value of the event encoded as a small non-negative number. Event numbers are ultimately used as an array index and therefore must be encoded as zero based sequential integers. The encoding is very tedious to keep maintained and again some tooling like the **pycca** program is helpful in this regard.

The `alloc` member is yet another part of *event-in-flight* detection (see page 68). We will discuss that more below when we discuss event dispatch. For now, this member of the ECB holds the value of the allocation counter for the instance that is the target of the event. So when an event is generated, a copy of the current value of the `alloc` member of the instance is stored in the ECB. I'm sure you can see where this is going.

The `eventType` member describes which of the three types of events this ECB is being used for. It also helps discriminate the anonymous union of `targetInst` and `targetClass`. For normal and polymorphic events, the events are directed at a class instance and therefore the `targetInst` member is used. For creation events, the events are directed, conceptually, at a class and the `targetClass` member is used.

The `srcInst` member records the instance pointer of the instance that generates the event. If the event is generated outside the context of an instance (*e.g.* in an domain operation), then this member value is set to `NULL`. The `srcInst` member also serves an important role in enforcing the rules for delayed events. More on that later.

The `delay` member is used in delayed event generation. We discuss delayed events in detail in section 4.3 (p. 39).

Finally, the `eventParameters` member holds parametric data for this event.

Note that there is no notion of priority contained in the ECB. Some software architectures queue events in a priority order. That is not supported here. Frankly, if you need event priorities to make your system work, then you need to revisit your design or look for a software architecture that supports multiple threads of execution.

4.2.3 Event Queuing

The last step in event generation is to queue the event. In this architecture, we do asynchronous event dispatch. This insures that the state actions are atomic. Since a queue is used, as a state action executes and generates an event, we know that event will not be dispatched until after the state action completes. Therefore, there is no danger of a long complicated chain of event dispatching cycling back around to alter the state of the instance or potentially modify some data that the state action accesses after generating the event. The guaranteed of atomic state action execution is very important. We now examine the code that performs the queueing.

This code is very conventional and I'm sure you seen it or something very much like it many times before. There are only a of operations on the queue. We need to determine the beginning and end of the queue as event are queue in order of generation.

26a ⟨*event queues* 26a⟩≡ (150 153 156) 26b ▷

```
static inline
struct mechecb *
eventQueueBegin(
    MechEcb iter)
{
    return iter->next ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechEcb` 24.

26b ⟨*event queues* 26a⟩+≡ (150 153 156) ◁ 26a 27a ▷

```
static inline
struct mechecb *
eventQueueEnd(
    MechEcb iter)
{
    return iter ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechEcb` 24.

We need to be able to determine if a queue is empty.

27a ⟨*event queues* 26a⟩+≡ (150 153 156) ◁26b 27b▷
```
static inline
bool
eventQueueEmpty(
    MechEcb iter)
{
    return iter->next == iter ;
}
```
Defines:
`inline`, used in chunk 101b.
Uses `MechEcb` 24.

And finally, we need to be able to insert and remove entries. Insertion places the ECB pointed to by `item` in the queue immediately before the ECB pointed to by `at`.

27b ⟨*event queues* 26a⟩+≡ (150 153 156) ◁27a 27c▷
```
static inline
void
eventQueueInsert(
    MechEcb item,
    MechEcb at)
{
    item->prev = at->prev ;
    item->next = at ;
    at->prev->next = item ;
    at->prev = item ;
}
```
Defines:
`inline`, used in chunk 101b.
Uses `MechEcb` 24.

Removal simply links around the ECB pointed to by `item`.

27c ⟨*event queues* 26a⟩+≡ (150 153 156) ◁27b 28b▷
```
static inline
void
eventQueueRemove(
    MechEcb item)
{
    item->prev->next = item->next ;
    item->next->prev = item->prev ;
}
```
Defines:
`inline`, used in chunk 101b.
Uses `MechEcb` 24.

Like all the other data structures, there is a storage pool for ECB's and we define a size for it here that can be overridden on the compiler command line. Sizing the pool for ECB's can be difficult. It must be worst case allocation as running out of ECB's is a fatal system error. The pool must be sized to account for the maximum number of events that can be in flight at the same time. This includes delayed events, since they can be considered to be slow flying events.

28a ⟨*constants* 11c⟩+≡ (149 152 155) ◁22 79a▷

```
#ifndef MECH_EVENTPOOLSIZE
#   define MECH_EVENTPOOLSIZE 10
#endif /* MECH_EVENTPOOLSIZE */
```

We must allocate the memory for the ECB storage. As usual, storage is just an array of structures.

28b ⟨*event queues* 26a⟩+≡ (150 153 156) ◁27c 28c▷

```
static struct mechecb mechECBPool[MECH_EVENTPOOLSIZE] ;
```

Defines:
`mechECBPool`, used in chunk 29.

There are four queues that are used to manage events. Note that all the queues are initialized to be empty, *i.e.* their `next` and `prev` members point back to the queue head itself.

28c ⟨*event queues* 26a⟩+≡ (150 153 156) ◁28b 29▷

```
static struct mechecb eventQueue ;
static struct mechecb delayedEventQueue ;
static struct mechecb freeEventQueue ;
```

Defines:
`delayedEventQueue`, used in chunks 29, 42, 46, 48, 50, and 52–56.
`eventQueue`, used in chunks 29, 35b, 36, 46, 54, 102c, 130b, and 132.
`freeEventQueue`, used in chunks 29, 30, and 93b.

The `eventQueue` queue holds all events waiting to be dispatched. The `delayedEventQueue` queue holds events that are to be delivered by the mechanisms at some time in the future. Finally, the `freeEventQueue` queue holds those ECB's that are not currently begin used. From the data structures and the semantics of the queuing, a given ECB can be on at most one of the queues at any time. Most of the time each ECB is on exactly one of the queues, but there are short times when ECB's are held in local variables or they can even be held in domain variables.

Since we have a pool of ECB's, we need some operations to manage the pool. We start with initialization. This places all the ECB's in the pool onto the free event queue.

29 ⟨*event queues* 26a⟩+≡ (150 153 156) ◁28c 30a▷

```
static void
mechEventInit(void)
{
    assert(MECH_EVENTPOOLSIZE >= 1) ;
    /*
     * Initialize the ECB used as the queue terminus.
     */
    eventQueue.next = eventQueue.prev = &eventQueue ;
    delayedEventQueue.next = delayedEventQueue.prev =
           &delayedEventQueue ;
    freeEventQueue.next = freeEventQueue.prev = &freeEventQueue ;
    /*
     * Place all the event control blocks on the free event
     * queue.  Allocation occurs from there.
     */
    for (MechEcb ecb = mechECBPool ;
            ecb < mechECBPool + MECH_EVENTPOOLSIZE ; ++ecb) {
        eventQueueInsert(ecb, &freeEventQueue) ;
    }
}
```

Uses `delayedEventQueue` 28c, `eventQueue` 28c, `freeEventQueue` 28c, `MechEcb` 24, and `mechECBPool` 28b.

Event allocation is just removing an ECB from the free list. *N.B.* that running out of Event Control Blocks is fatal.

30a ⟨*event queues* 26a⟩+≡ (150 153 156) ◃29 30b▹

```
static inline
MechEcb
mechEventAlloc(void)
{
    if (eventQueueEmpty(&freeEventQueue)) {
        mechFatalError(mechNoECB) ;
    }

    MechEcb ecb = freeEventQueue.next ;
    eventQueueRemove(ecb) ;
    ecb->referenceCount = 0 ;
    return ecb ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `freeEventQueue` 28c and `MechEcb` 24.

Finally, deletion returns ECB's to the free queue.

30b ⟨*event queues* 26a⟩+≡ (150 153 156) ◃30a

```
static void
mechEventDelete(
    MechEcb ecb)
{
    assert(ecb != NULL) ;

    if (ecb->referenceCount <= 1) {
        eventQueueInsert(ecb, &freeEventQueue) ;
    } else {
        --ecb->referenceCount ;
    }
}
```

Uses `freeEventQueue` 28c and `MechEcb` 24.

Note that ECB's are returned to the free queue only when their reference count goes to zero. The test, `if (ecb->referenceCount <= 1)`, is used since the `referenceCount` member is unsigned and we want to free the ECB if its reference count is either 0 or 1 when it is passed to this function. This allows an invocation of `mechEventAlloc` followed immediately by an invocation of `mechEventDelete` to properly free the ECB. Blindly decrementing the `referenceCount` will cause it to roll over when it is 0.

4.2.4 Interface for Event Generation

Finally, we can get to functions that the mechanisms provide for a state action to generate an event.

One step in generating an event is to fill in the ECB members. Much of that step is common among the three types of events and we can factor that into a constructor type function.

31a ⟨*event generate* 31a⟩≡ (150 153 156) 32a ▷

```
static inline
MechEcb
mechEventCtor(
    EventCode event,
    MechEventType type,
    MechInstance targetInst,
    MechInstance srcInst)
{
    MechEcb ecb = mechEventAlloc() ;

    ecb->eventNumber = event ;
    ecb->eventType = type ;
    ecb->instOrClass.targetInst = targetInst ;
    ecb->srcInst = srcInst ;

    return ecb ;
}
```

Defines:
  `inline`, used in chunk 101b.
Uses `MechEcb` 24, `MechEventType` 21, and `MechInstance` 10.

First we consider the step of obtaining an ECB. For normal events the `mechEventNew` function is provided.

31b ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁ 18b 32b ▷

```
extern MechEcb mechEventNew(
    EventCode event,
    MechInstance targetInst,
    MechInstance srcInst) ;
```

Uses `MechEcb` 24 and `MechInstance` 10.

`event` The number of the event.

`targetInst` A pointer to the instance structure that is to receive the event.

`srcInst` A pointer to the instance structure that is sending the event. Events generated outside of a class instance may set this argument to `NULL`.

This function returns an ECB for a normal event that has all its internal members initialized properly.

32a ⟨*event generate* 31a⟩+≡ (150 153 156) ◁31a 33a▷

```
MechEcb
mechEventNew(
    EventCode event,
    MechInstance targetInst,
    MechInstance srcInst)
{
    assert(targetInst != NULL) ;
    assert(targetInst->alloc != 0) ;
    assert(targetInst->instClass != NULL) ;
    assert(targetInst->instClass->odb != NULL) ;

    MechEcb ecb = mechEventCtor(event, NormalEvent, targetInst,
            srcInst) ;
    /*
     * Take a copy of the alloc member for event-in-flight
     * detection.
     */
    ecb->alloc = targetInst->alloc ;
    return ecb ;
}
```

Uses `MechEcb` 24 and `MechInstance` 10.

Note the assignment of the the `alloc` member from the instance into the `alloc` member of the ECB. This is yet another part of the *event-in-flight* detection (see page 68).

As you might expect, there is a corresponding function for each of the other two event types.

32b ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁31b 33b▷

```
extern MechEcb
mechPolyEventNew(
    EventCode event,
    MechInstance targetInst,
    MechInstance srcInst) ;
```

Uses `MechEcb` 24 and `MechInstance` 10.

`event` The number of the event. *N.B.* the event number here is the event number of the polymorphic event as encoded for a class that serves as a supertype in a generalization hierarchy.

`targetInst` A pointer to the instance structure that is to receive the event.

`srcInst` A pointer to the instance structure that is sending the event. Events generated outside of a class instance may set this argument to `NULL`.

33a ⟨*event generate* 31a⟩+≡ (150 153 156) ◁32a 34a▷

```
MechEcb
mechPolyEventNew(
    EventCode event,
    MechInstance targetInst,
    MechInstance srcInst)
{
    assert(targetInst != NULL) ;
    assert(targetInst->instClass != NULL) ;
    assert(targetInst->instClass->pdb != NULL) ;

    MechEcb ecb = mechEventCtor(event, PolymorphicEvent,
            targetInst, srcInst) ;
    return ecb ;
}
```

Uses `MechEcb` 24 and `MechInstance` 10.

Note that the `alloc` member is not assigned to for polymorphic events. This is because polymorphic events are never delivered to an instance. They are mapped at run time to a normal event which is delivered to the instance.

33b ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁32b 35a▷

```
extern MechEcb
mechCreationEventNew(
    EventCode event,
    MechClass targetClass,
    MechInstance srcInst) ;
```

Uses `MechEcb` 24 and `MechInstance` 10.

34a ⟨*event generate* 31a⟩+≡ (150 153 156) ◁33a 34b▷
```
MechEcb
mechCreationEventNew(
    EventCode event,
    MechClass targetClass,
    MechInstance srcInst)
{
    assert(targetClass != NULL) ;
    assert(targetClass->iab != NULL) ;
    assert(targetClass->odb != NULL) ;

    MechEcb ecb = mechEventAlloc() ;

    ecb->eventNumber = event ;
    ecb->eventType = CreationEvent ;
    ecb->instOrClass.targetClass = targetClass ;
    ecb->srcInst = srcInst ;
    ecb->alloc = 0 ;

    return ecb ;
}
```
Uses `MechEcb` 24 and `MechInstance` 10.

Creation events are different enough that the `mechEventCtor` function is not that useful and the initialization is more easily accomplished directly.

As discussed above, ECB's are reference counted. Internally, the mechanisms use the function below to increment the reference count for an ECB before it is inserted into a queue.

34b ⟨*event generate* 31a⟩+≡ (150 153 156) ◁34a 35b▷
```
static inline
RefCount
mechEventIncrRef(
    MechEcb ecb)
{
    return ++ecb->referenceCount ;
}
```
Defines:
  `inline`, used in chunk 101b.
Uses `MechEcb` 24.

There is no corresponding function to decrement the reference count as `mechEventDelete` serves that role.

Once you have obtained an ECB initialized for the proper type of the event, then you need to fill in any event parameter data. Frequently, there are none. Then the ECB is ready to be placed on a queue. As mentioned before, there is a distinction between events an instance sends to itself and those that an instance sends to a different instance. Self directed events are placed on the front of the event queue so that they are dispatched in preference to the non-self directed events. It is important to place the ECB in the correct position on the event queue, but the mechanisms provide no guarantees for this other than the assistance of an assertion.

To queue a non-self directed event, the function `mechEventPost` is invoked.

35a ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁33b 35c▷

```
extern void mechEventPost(MechEcb ecb) ;
```

Uses `MechEcb` 24.

`ecb` A pointer to an event control block (ECB). The ECB must be the return value from `mechEventNew`, `mechPolyEventNew` or `mechCreationEventNew`.

The code is very straight forward. The reference count of the ECB is incremented and it is queued to the rear of the event queue.

35b ⟨*event generate* 31a⟩+≡ (150 153 156) ◁34b 36▷

```
void
mechEventPost(
    MechEcb ecb)
{
    mechEventIncrRef(ecb) ;
    eventQueueInsert(ecb, &eventQueue) ;
}
```

Uses `eventQueue` 28c and `MechEcb` 24.

To queue a self directed event, the function `mechEventPostSelf` is invoked.

35c ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁35a 41▷

```
extern void mechEventPostSelf(MechEcb ecb) ;
```

Uses `MechEcb` 24.

`ecb` A pointer to an event control block (ECB). The ECB must be the return value from `mechEventNew`, `mechPolyEventNew` or `mechCreationEventNew`.

The complication that arises when posting self directed events comes from finding where in the queue to insert the ECB. At first glance, it would seem that we need only put the event on the front of the event queue. However, should a state action generate two self directed events, simply placing them on the front of the event queue would result in the events being delivered in the reverse of the order that they were generated. It's an unusual case, but it is simple enough to cure by finding the first event on the event queue where the source instance and the target instance are different and performing the insertion of the ECB there.

36 ⟨*event generate* 31a⟩+≡ (150 153 156) ◁35b 93b▷

```
void
mechEventPostSelf(
    MechEcb ecb)
{
    assert(ecb->instOrClass.targetInst == ecb->srcInst) ;
    /*
     * Find the first event that is not self-directed.
     */
    MechEcb iter ;
    for (iter = eventQueueBegin(&eventQueue) ;
            iter != eventQueueEnd(&eventQueue) ;
            iter = iter->next) {
        if (iter->srcInst != iter->instOrClass.targetInst) {
            break ;
        }
    }

    mechEventIncrRef(ecb) ;
    eventQueueInsert(ecb, iter) ;
}
```

Uses `eventQueue` 28c and `MechEcb` 24.

Because the case where there are no event parameters is so common, we provide a set of convenience functions that can be used to allocate and queue an ECB for the various types of events and queues. In many cases this simplifies the coding significantly.

37a ⟨*inline functions* 37a⟩≡ (149 152 155) 37b ▷

```
static inline
void
mechEventGenerate(
    EventCode event,
    MechInstance target,
    MechInstance source)
{
    mechEventPost(mechEventNew(event, target, source)) ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechInstance` 10.

37b ⟨*inline functions* 37a⟩+≡ (149 152 155) ◁ 37a 37c ▷

```
static inline
void
mechEventGenerateToSelf(
    EventCode event,
    MechInstance target)
{
    mechEventPostSelf(mechEventNew(event, target, target)) ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechInstance` 10.

37c ⟨*inline functions* 37a⟩+≡ (149 152 155) ◁ 37b 38 ▷

```
static inline
void
mechEventGeneratePolymorphic(
    EventCode event,
    MechInstance target,
    MechInstance source)
{
    mechEventPost(mechPolyEventNew(event, target, source)) ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechInstance` 10.

38 ⟨*inline functions* 37a⟩+≡ (149 152 155) ◁37c 44▷

```
static inline
void
mechEventGenerateCreation(
    EventCode event,
    MechClass targetClass,
    MechInstance source)
{
    mechEventPost(mechCreationEventNew(event, targetClass,
            source)) ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechInstance` 10.

4.3 Delayed Events

The concept of a delayed event is one where instead of the event begin posted to the event queue immediately, the mechanisms are requested to post the event at some time in the future. This implies that the mechanisms have access to some type of timing facility by which they can know that a given amount of time has elapsed and this implies that the mechanisms will hold on to the ECB until that future time has arrived.

There is one significant XUML rule associated with delayed events. There can be only one outstanding delayed event of a given event type between any sending / receiving pair of instances (which may be the same instance). This is another way of stating that delayed events are identified by their event name (or numerical encoding), the target instance and the source instance. There are a number of ways to interpret an attempt to generate what amounts to a duplicate delayed event. It could be considered an error, but that is inconvenient and goes against the grain of our attempts to minimize run-time errors. So the mechanisms regard an attempt to generate a delayed event of the same name between the same sending and receiving pair as a request to cancel the original event and create the new one at its newly given time. This turns out to be very convenient in practice, eliminating the need to perform checks. Cancelling and reinstating a new event turns out to be what is required in most circumstances.

To understand delayed events, it is necessary to understand the way the delayed event queue is maintained. The mechanisms have a delayed event queue where ECB's are placed awaiting to be posted. In servicing the delayed events, we are particularly trying to avoid doing any periodic computation. For example, we could treat the delayed event queue as a simple list and wake up periodically and run down the list decrementing time values and checking if any events have expired. Such a scheme is easy to implement, but in highly embedded and power sensitive application, periodic activity of this type is wasteful and deemed inappropriate.

In this implementation, we keep the delayed event queue in time relative order. Consider the following diagram.

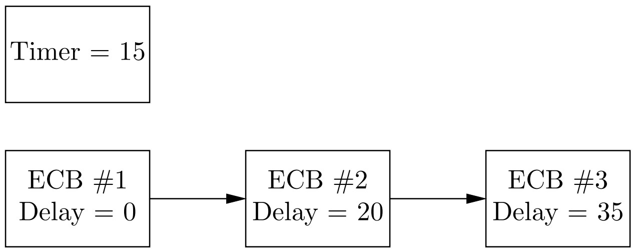

Figure 2: Delayed Event Ordering

Here we show a conceptual timer that is counting down from 15 counts to zero. When the timer gets to zero, ECB #1 will have expired (because its delay value, relative to the timer, is 0) and is placed on the event queue. The timer is then loaded with the delay value from ECB #2, namely 20. In this way, ECB #2 will have waited for the total time of all its predecessors plus its own delay time before being placed on the event queue.

Now consider state of the delayed event queue as originally shown in figure 2. If at this time a delayed event request, call it ECB # 4, arrives for 45, then the state of the queue after the request is inserted is shown in figure 3.

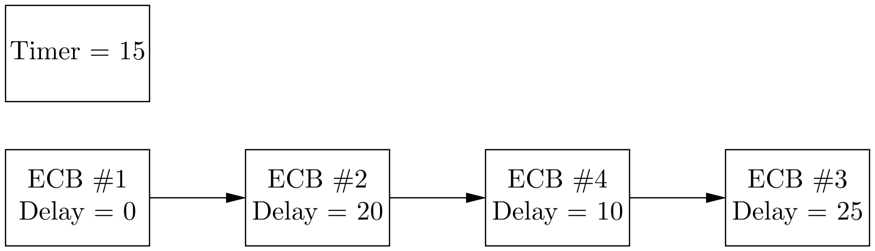

Figure 3: Delayed Event Ordering After Insertion

So ECB #4 will expire in 45 ticks because it will have waited for 15 ticks from ECB #1, 20 ticks from ECB #2 and 10 ticks for itself. Note also, that ECB #3 has had its delay time decreased by 10 ticks. This means that ECB #3 will expire at the same time as it would have had the insertion not taken place (*i.e.* 15 + 20 + 10 + 25 equals 15 + 20 + 35).

This design meets two important criteria; only a single source of timing is used and there is no periodic execution activity. The cost of meeting these criteria is the price paid to find the appropriate place in the delayed event queue when a delayed event is requested.

There are three functions supplied for dealing with delayed events:

- Post a delayed event.
- Cancel a delayed event.
- Query the remaining time for a delayed event.

While we are talking about time, the question of what time units must be answered. Time is specified in units of milliseconds.

40 ⟨*base types* 11a⟩+≡ (149 152 155) ◁23c 64a▷

```
typedef unsigned long int MechDelayTime ;
```

Defines:
`MechDelayTime`, used in chunks 24, 41, 42, 44, 45a, 47, 48, 56, 58, 59, 105, 107–109, 125, 126, 138, 143, 144, and 150.

4.3.1 mechEventPostDelay

Posting a delayed event is just like posting a normal event, except that you must supply the number of milliseconds from now when the event is to be delivered.

41 ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁35c 45b▷

```
extern void
mechEventPostDelay(
    MechEcb ecb,
    MechDelayTime time) ;
```

Uses `MechDelayTime` 40 and `MechEcb` 24.

`ecb` A pointer to an event control block (ECB). The ECB must be the return value from `mechEventNew()`, `mechPolyEventNew()` or `mechCreationEventNew()`.

`time` The number of milliseconds in the future when the ECB pointed to by `ecb` is posted to the event queue. Note that this is the minimum number of milliseconds and more time may elapse before the event is actually posted.

42 ⟨*delayed event queue* 42⟩≡ (150 153 156) 46 ▷

```
void
mechEventPostDelay(
    MechEcb ecb,
    MechDelayTime time)
{
    assert(ecb != NULL) ;
    /*
     * A delay time of 0 is valid, and the event will be
     * queued immediately.
     */
    if (time == 0) {
        mechEventPost(ecb) ;
        return ;
    }
    ecb->delay = mechMsecToTicks(time) ;
    /*
     * Stop the timing queue so we may examine it.
     */
    stopDelayedQueueTiming() ;
    /*
     * If the event already exists, remove it.
     */
    MechEcb prevEvent = findEvent(&delayedEventQueue,
            ecb->srcInst, ecb->instOrClass.targetInst,
            ecb->eventNumber) ;
    if (prevEvent) {
        removeFromDelayedQueue(prevEvent) ;
    }
    /*
     * Insert the new event.
     */
    insertIntoDelayedQueue(ecb) ;
    assert(!eventQueueEmpty(&delayedEventQueue)) ;
    /*
     * Start the timer to expire for the first event
     * on the queue.
     */
    startDelayedQueueTiming() ;
}
```

Uses `delayedEventQueue` 28c, `MechDelayTime` 40, and `MechEcb` 24.

There are six main actions of this function.

1. First, a quick check is done to see if the delay time is zero. If is it, then the event is posted immediately. Delayed events are always posted as non-self directed events. Using a zero delay time is a "trick" that provides a means for a thread of control the *yield* up the processor by posting a delayed event directed at itself. This will place the event at the end of the event queue allowing all other pending events to be consumed before dispatching back to the yielding instance. The mechanism is crude, but sometimes necessary when one is forced to use the processor for unbounded computations (*e.g.* copying memory from one place to another).
2. Next, we convert the delay value from millisecond units to units of *ticks*. A tick is platform specific. Computers don't typically keep time at the lowest level in conventional human units. However, we would like to run the delayed event queue in system specific units to avoid as much unnecessary conversion as we can. This conversion will be described below for each supported platform.
3. Next, we stop the timing of the delayed event queue. More on this later but the goal of stopping the delayed event queue timing is to freeze the state of the queue so that we may operate on it.
4. Next we attempt to determine if there is already an event matching the one begin posted. This enforces the rule about not having two delayed events of the same type between the same sending / receiving pair. If one is found then it is removed.
5. Assured of no duplicates, the new event can be inserted into the timing queue. We will see how that happens below.
6. Finally, the timing of the delayed queue is started.

It is also convenient to define delayed event functions that generate events that have no supplemental event data.

44 ⟨*inline functions* 37a⟩+≡ (149 152 155) ◁38 45a▷

```
static inline
void
mechEventGenerateDelayed(
    EventCode event,
    MechInstance target,
    MechInstance source,
    MechDelayTime delay)
{
    mechEventPostDelay(mechEventNew(event, target, source),
            delay) ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechDelayTime` 40 and `MechInstance` 10.

45a ⟨*inline functions* 37a⟩+≡ (149 152 155) ◁44
```
static inline
void
mechEventGenerateDelayedToSelf(
    EventCode event,
    MechInstance target,
    MechDelayTime delay)
{
    mechEventPostDelay(mechEventNew(event, target, target),
            delay) ;
}
```
Defines:
`inline`, used in chunk 101b.
Uses `MechDelayTime` 40 and `MechInstance` 10.

4.3.2 mechEventDelayCancel

Cancelling a delayed event is one of the more complicated delayed event operations. We must account for the various places where a delayed event may be queued.

A delayed event may be in one of two places:

1. In the delayed event queue awaiting either waiting for its time to expire or having already been marked as expired.
2. In the event queue awaiting dispatch.

We will have more to say about event dispatch below, but it is possible to try to cancel an event after its time has expired but before it has been delivered to the target instance. The mechanisms make the guarantee that after invoking `mechEventDelayCancel` the application can be assured that the event will *not* be delivered until such time as it is posted again. Note that it is possible to attempt to cancel a delayed event after it has already been delivered. This is not an error. Unfortunately, the mechanisms cannot turn time backwards.

45b ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁41 47▷
```
extern void
mechEventDelayCancel(
    EventCode event,
    MechInstance targetInst,
    MechInstance srcInst) ;
```
Uses `MechInstance` 10.

`event` The number of the event.

`targetInst` A pointer to the instance structure that is to receive the event.

`srcInst` A pointer to the instance structure that is sending the event. Events generated outside of a class instance may set this argument to `NULL`.

46 ⟨*delayed event queue* 42⟩+≡ (150 153 156) ◁42 48▷

```
void
mechEventDelayCancel(
    EventCode event,
    MechInstance targetInst,
    MechInstance srcInst)
{
    assert(targetInst != NULL) ;
    /*
     * Stop delayed queue so that we may examine it.
     */
    stopDelayedQueueTiming() ;
    /*
     * Search for the event in the delayed event queue.
     */
    MechEcb foundEvent = findEvent(&delayedEventQueue,
            srcInst, targetInst, event) ;
    if (foundEvent) {
        /*
         * Removing from the delayed queue requires
         * additional processing of the delay times.
         */
        removeFromDelayedQueue(foundEvent) ;
    } else {
        /*
         * If the event is not in the delayed queue, then
         * search the event queue. The timer could have
         * expired and the event placed in the queue.
         */
        foundEvent = findEvent(&eventQueue, srcInst,
                targetInst, event) ;
        if (foundEvent) {
            eventQueueRemove(foundEvent) ;
            mechEventDelete(foundEvent) ;
        }
        /*
         * We can get here, without finding the event in the
         * delayed queue or the event queue.
         * That’s okay, it just amounts to an expensive
         * no-op and implies that the event has expired,
         * was queued and has already been dispatched or
         * had never been generated at all.
         */
    }
    startDelayedQueueTiming() ;
}
```

Uses `delayedEventQueue` 28c, `eventQueue` 28c, `MechEcb` 24, and `MechInstance` 10.

Like all operations dealing with the delayed event queue, we must first put the queue in a state that we can examine it without asynchronous timing services modifying its state. Then we search the delayed event queue for the event to cancel. The only complication in the implementation is the need to search the event queue should the ECB have already been expired off of the delayed queue.

4.3.3 mechEventDelayRemaining

The remaining provided operation on delayed events is to query the amount of time remaining for a particular delayed event. Since we hold the delayed events in sorted order of time differences, the task of determining the amount of remaining time involves traversing the queue and summing the time increments of all the events in front of the event of interest. The only special case here is what to do if we don't find the delayed event at all. In that case, zero is returned.

47 ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁45b 58▷

```
extern MechDelayTime
mechEventDelayRemaining(
    EventCode event,
    MechInstance targetInst,
    MechInstance srcInst) ;
```

Uses `MechDelayTime` 40 and `MechInstance` 10.

`event` The number of the event.

`targetInst` A pointer to the instance structure that is to receive the event.

`srcInst` A pointer to the instance structure that is sending the event. Events generated outside of a class instance may set this argument to `NULL`.

The algorithm for computing the remaining time is to simply walk the delayed event queue from the beginning, summing up the set of time delays until we reach the ECB of delayed event.

48 ⟨*delayed event queue* 42⟩+≡ (150 153 156) ◁46

```
MechDelayTime
mechEventDelayRemaining(
    EventCode event,
    MechInstance targetInst,
    MechInstance srcInst)
{
    assert(targetInst != NULL) ;

    stopDelayedQueueTiming() ;
    /*
     * Iterate through the delayed event time and sum all
     * the delay times to give the total amount of time
     * remaining for the found event.
     */
    MechDelayTime remain = 0 ;
    MechEcb iter ;
    for (iter = eventQueueBegin(&delayedEventQueue) ;
            iter != eventQueueEnd(&delayedEventQueue) ;
            iter = iter->next) {
        remain += iter->delay ;
        if (iter->srcInst == srcInst &&
                iter->instOrClass.targetInst == targetInst &&
                iter->eventNumber == event) {
            break ;
        }
    }
    startDelayedQueueTiming() ;
    /*
     * Return the amount of time remaining for the event.
     * If we didn't find the event, the just return 0.
     */
    return iter == eventQueueEnd(&delayedEventQueue) ?
            0 : mechTicksToMsec(remain) ;
}
```

Uses `delayedEventQueue` 28c, `MechDelayTime` 40, `MechEcb` 24, and `MechInstance` 10.

Note that zero is returned if we did not find the event on the delayed queue. Returning zero does *not* tell us if the event has already been dispatched or might be still in flight on the event queue.

4.3.4 Delayed Event Queue Operations

In this section we will look at the operations on the delayed event queue itself. As we have seen above, the events in the delayed event queue are ordered by relative time. Inserting and removing from the queue must keep the ECB's in time relative order.

First, we need some way to find an event in the delayed queue. Turns out we will end up needing to search an arbitrary queue and so we define a function to do just that.

49 ⟨*delayed event helper* 49⟩≡ (150 153 156) 50▷

```
static MechEcb
findEvent(
    MechEcb queue,
    MechInstance srcInst,
    MechInstance targetInst,
    EventCode event)
{
    /*
     * Simple iteration through the list of events
     * in the queue.
     */
    for (MechEcb iter = eventQueueBegin(queue) ;
            iter != eventQueueEnd(queue) ;
            iter = iter->next) {
        if (iter->srcInst == srcInst &&
                iter->instOrClass.targetInst == targetInst &&
                iter->eventNumber == event) {
            return iter ;
        }
    }
    return NULL ;
}
```

Uses `MechEcb` 24 and `MechInstance` 10.

Now we present inserting into the delayed event queue.

50 ⟨*delayed event helper* 49⟩+≡ (150 153 156) ◁49 51▷

```
static void
insertIntoDelayedQueue(
    MechEcb ecb)
{
    /*
     * We walk down the queue to find the correct slot.
     * That slot is the first place in the queue where our
     * delay value is less than the delay value at that
     * place in the queue.  As we walk the queue, we
     * subtract the delay value of each entry we pass since
     * that entry will have expired before the one being
     * inserted.
     */
    MechEcb iter ;
    for (iter = eventQueueBegin(&delayedEventQueue) ;
            iter != eventQueueEnd(&delayedEventQueue) ;
            iter = iter->next) {
        /*
         * By keeping this comparison to be strictly less
         * than, we preserve the order of event dispatch to
         * match that of event generation.
         */
        if (ecb->delay < iter->delay) {
            /*
             * We are going to insert before the entry
             * pointed to by "iter". Therefore, we need to
             * decrease its delay value by the amount of
             * time we will cause to elapse by expiring the
             * entry we are about to insert.
             */
            iter->delay -= ecb->delay ;
            break ;
        } else {
            ecb->delay -= iter->delay ;
        }
    }
    /*
     * At this point we have found our place in the queue.
     * Either we are between entries or this delay was
     * longer than the cumulative delays of all the ECB's
     * in the queue. Insert the ECB.
     */
    eventQueueInsert(ecb, iter) ;
```

```
        /*
         * Since we have stored a reference to the ECB we do
         * the bookkeeping.
         */
        mechEventIncrRef(ecb) ;
    }
```

Uses `delayedEventQueue` 28c and `MechEcb` 24.

When the timer expires there are several ways to deal with the events that need to be dispatched. One temptation would be to simply sync to the background and let code run there to remove expired events from the delayed event queue and restart the timer. Unfortunately, that would introduce considerable jitter into the timing. We need to deal with the expired events and restart the timer quickly and in a deterministic way. This design chooses to mark the events in the delayed event queue as expired and that provides a means of knowing which events in the delayed queue are active. Since a delay value of 0 is meaningful, we use a delay value of the maximum for a delay time as the expired marker.

51 ⟨*delayed event helper* 49⟩+≡ (150 153 156) ◁50 52▷

```
    #define MECH_DELAY_EXPIRED  UINT32_MAX
```

Defines:
`MECH_DELAY_EXPIRED`, used in chunks 52–54.

Removing an entry from the queue is much simpler. The only job here is to account for the time that the entry would have consumed had it been left in the queue. That time must be added to its next neighbor (if there is one). Note that we must also be careful to ignore any expired events.

52 ⟨*delayed event helper* 49⟩+≡ (150 153 156) ◁51 53▷

```
static void
removeFromDelayedQueue(
    MechEcb ecb)
{
    /*
     * If we are not at the end of the queue, all the delay
     * from the removed entry is accumulated on the next
     * entry in the queue.
     */
    if (!(ecb->delay == MECH_DELAY_EXPIRED ||
            ecb->next == eventQueueEnd(&delayedEventQueue))) {
        ecb->next->delay += ecb->delay ;
    }
    /*
     * Remove the ECB from the delayed queue.
     */
    eventQueueRemove(ecb) ;
    /*
     * Return the ECB back to the pool.
     */
    mechEventDelete(ecb) ;
}
```

Uses `delayedEventQueue` 28c, `MECH_DELAY_EXPIRED` 51, and `MechEcb` 24.

The `expireDelayedEvents` function is run to traverse the delayed event queue and mark any events that have a delay value of 0 as expired. It is used both in the delayed event timer service as well as in the background processing of the delayed event queue. The return value is pointer to an ECB. This will be `NULL` if there are no other events in the delayed queue that require timing. Otherwise, the returned ECB pointer will reference the first unexpired, non-zero delay event. The semantics are a bit strained, but the function is used for two distinct purposes.

53 ⟨*delayed event helper* 49⟩+≡ (150 153 156) ◁52 54▷

```
static MechEcb
expireDelayedEvents(void)
{
    /*
     * Iterate along the delayed event queue.
     */
    for (MechEcb iter = eventQueueBegin(&delayedEventQueue) ;
            iter != eventQueueEnd(&delayedEventQueue) ;
            iter = iter->next) {
        if (iter->delay == 0) {
            /*
             * Mark all the events that have zero delay time
             * as expired.
             */
            iter->delay = MECH_DELAY_EXPIRED ;
        } else if (iter->delay != MECH_DELAY_EXPIRED) {
            /*
             * Stop at the first non-zero delay time.  This
             * marks the boundary of events that need
             * additional delay time.  The first such event
             * is the next amount of time to delay.
             */
            return iter ;
        }
        /*
         * else ... Skip any events that might already be
         * expired.
         */
    }
    /*
     * We have run the queue without finding an unexpired
     * event.
     */
    return NULL ;
}
```

Uses `delayedEventQueue` 28c, `MECH_DELAY_EXPIRED` 51, and `MechEcb` 24.

The function below is then run in the background to re-queue the expired events.

54 ⟨*delayed event helper* 49⟩+≡ (150 153 156) ◁53 55▷

```
static void
transferExpiredEvents(void)
{
    /*
     * Iterate through the delayed event queue looking for
     * those entries that have been marked as expired.
     */
    for (MechEcb iter = eventQueueBegin(&delayedEventQueue) ;
            iter != eventQueueEnd(&delayedEventQueue) &&
            iter->delay == MECH_DELAY_EXPIRED ; ) {
        /*
         * Advance the iterator, because we are about to
         * invalidate it by removing the entry from the
         * queue.
         */
        MechEcb ecb = iter ;
        iter = iter->next ;

        /*
         * Remove the ECB from the delayed queue and insert
         * it into event queue for dispatch.
         */
        eventQueueRemove(ecb) ;
        eventQueueInsert(ecb, &eventQueue) ;
        assert(ecb->referenceCount != 0) ;
    }
}
```

Uses `delayedEventQueue` 28c, `eventQueue` 28c, `MECH_DELAY_EXPIRED` 51, and `MechEcb` 24.

The concept of starting the delayed event queue timing is associated with starting the timing resource with the delay time of the event on the front of the delayed queue. We move the delay value from the head of the delayed event queue into the timer and zero the delay member.

55 ⟨*delayed event helper* 49⟩+≡ (150 153 156) ◁54 56▷

```
static void
startDelayedQueueTiming(void)
{
    if (!eventQueueEmpty(&delayedEventQueue)) {
        MechEcb ecb = eventQueueBegin(&delayedEventQueue) ;
        assert(ecb->delay != 0) ;
        sysTimerStart(ecb->delay) ;
        ecb->delay = 0 ;
    }
}
```

Uses `delayedEventQueue` 28c and `MechEcb` 24.

An analogous operation is needed to stop the queue timing. Any remaining time is set back into the first entry on the queue. This puts the queue into a state where ECB's can be inserted, deleted or summed to find the time remaining for an event.

56 ⟨*delayed event helper* 49⟩+≡ (150 153 156) ◁55

```
static void
stopDelayedQueueTiming(void)
{
    /*
     * Avoid the whole thing if there is nothing in the
     * delayed event queue.
     */
    if (!eventQueueEmpty(&delayedEventQueue)) {
        /*
         * Stop the timer, obtaining the residual time.
         */
        MechDelayTime remain = sysTimerStop() ;
        /*
         * There are two cases here. It is possible for the
         * remaining time returned from sysTimerStop() to be
         * zero. This can happen if the physical timing
         * resource (which might be running asynchronously
         * to the processor) happens to expire within a
         * single tick as we are stopping it.
         */
        if (remain == 0) {
            /*
             * Since the timer has expired we must mark any
             * events with a zero delay time value as
             * expired and, since we are running in the
             * background here, transfer the expired events
             * to be dispatched.
             */
            expireDelayedEvents() ;
            transferExpiredEvents() ;
            /*
             * At this point, either the delayed event queue
             * is empty, or the event at the head of the
             * queue has a non-zero delay time.
             */
        } else {
            /*
             * It is possible that the timing resource
             * expired and its interrupt service ran just
             * before we could get the timer stopped. That
```

```
              * would mean that there are expired events on
              * the delayed queue at this point and we need
              * to transfer them off the delayed queue to be
              * dispatched.
              */
             transferExpiredEvents() ;
             /*
              * If any events expired, the delayed event
              * queue might now be empty. However, if the
              * queue is not empty, we must make sure the
              * entry at the head preserves the remaining
              * amount of time that needs to elapse.
              */
             if (!eventQueueEmpty(&delayedEventQueue)) {
                 MechEcb ecb = eventQueueBegin(&delayedEventQueue) ;
                 assert(ecb->delay == 0) ;
                 ecb->delay = remain ;
             }
         }
     }
 }
```

Uses `delayedEventQueue` 28c, `MechDelayTime` 40, and `MechEcb` 24.

Conceptually, starting and stopping the queue timing moves the time value of the first ECB on the delay queue into and out of the real timing resource (whatever that may be). So when the the delayed event queue timing is running, at least the first ECB on the queue will have a zero delay time. When it is stopped, we must insure that the first queued ECB has a non-zero delay time.

There is a race condition between the background code executing `stopDelayedQueueTiming()` and the timer services that may run asynchronously as an interrupt. After the return from `sysTimerStop()` the timer will have been stopped and will not cause any interrupt. However, at any time previous to that the timer interrupt might have gone off and expired one or more events. Although a sync function will have been posted, it will *not* have had an opportunity to run. So we must make sure to transfer any expired events to the event queue. This insures that the event generation does not get out of order.

The other complication here is that we might stop the timer just the instant before it expired. In this case the underlying timer services might think that there is no time left on the timer, but the timer interrupt did no actually occur. This is the case where we just win the race as opposed to the previous situation where we just lost the race. So, just in case, we transfer any events on the delay queue that happen to show zero delay times to the event queue.

4.3.5 Expired Events in the Delayed Event Queue

One could simply expire the timer, notify the background via a sync function and then allow background processing to remove the expired entries from the delayed event queue directly to the main event queue for dispatching. After the expired entries are queued, then the timer would be restarted for the next expiration time.

Such a design is subject to problematic, unpredictable timing skew. After the timer source has expired and posted the synchronization request, the background may not execute immediately. Either a currently executing state action or a previously queued sync function would delay the transfer of expired events from the delayed queue to the event queue. If there if another delayed event to expire, we would like to start that expiration without inserting some unpredictable time in between. Essentially, we would like to overlap the timing of the next delayed event with the time it takes to complete the current activities and execute the timer sync request.

The question then is how to signify that an event in the delayed event queue has expired. We mark its `delay` member with a special value to indicated that it has expired but not yet been removed from the delayed event queue. This allows us to keep track of the state of the ECB in the delayed event queue and still find the next timer interval during the timer interrupt service routine.

The `mechTimerExpireService` function is provided for the asynchronous timer execution to call that performs task of expiring events and finding the next time that needs to be placed in the timer.

58 ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁47 91b▷

```
/*
```

```
  * Must be invoked from interrupt service level only!
  */
 extern MechDelayTime mechTimerExpireService(void) ;
```

Uses `MechDelayTime` 40.

59 ⟨*timer service* 59⟩≡ (150 153 156)

```
 MechDelayTime
 mechTimerExpireService(void)
 {
     MechEcb unexpired ;
     MechDelayTime nextTime ;
     /*
      * Sync to the background to request the expired events
      * be transferred to the event queue.
      */
     mechSyncRequest(mechExpiredEventService) ;
     /*
      * Mark the delayed events as expired, returning a
      * pointer to the first unexpired event.
      */
     unexpired = expireDelayedEvents() ;
     if (unexpired) {
         /*
          * If there is an unexpired event, then its delay
          * time is the next time to expire. We return that
          * time and zero out the delay time.
          */
         assert(unexpired->delay != 0) ;
         nextTime = unexpired->delay ;
         unexpired->delay = 0 ;
     } else {
         /*
          * Otherwise, there is nothing else to time.
          */
         nextTime = 0 ;
     }

     return nextTime ;
 }
```

Uses `MechDelayTime` 40 and `MechEcb` 24.

There are two important aspects of this function.

1. The function assumes it is called from interrupt service level, *i.e.* it assumes that it may touch the delayed event queue with impunity since it cannot be interrupted by other code that might manipulate the state of those queues.

2. The function returns the next time to expire. The caller is responsible for placing that time into the timer facility if it is non-zero. If the returned time is zero, then the timer facility is not currently needed and should remain stopped.

Notice that the function `mechExpiredEventService` is queued as as sync function to perform the last step of moving the expired events to the event queue. The mechanisms use their own internal facilities to post this function as a sync function. This insures that delayed event posting is no different than any other interrupt synchronization and preserves the notion that the event queues are not accessed by asynchronous execution.

60 ⟨*delayed event service* 60⟩≡ (150 153 156)

```
static void
mechExpiredEventService(
    SyncParamRef params) /* Not used */
{
    /*
     * Since the expired event queue is accessed here,
     * we must make sure that the timer interrupt does not
     * go off.
     */
    sysTimerMask() ;
    transferExpiredEvents() ;
    sysTimerUnmask() ;
}
```

Uses `SyncParamRef` 97a.

4.3.6 Timing Considerations

With timing being such a common activity in programming, there are very many system specific situations that arise in obtaining timing services on any particular platform. When running on top of an operating system, it will provide the necessary timing services. Unfortunately, the interface to those services varies from OS to OS. On bare metal platforms, generally you will have to get timer peripherals and interrupts involved. We will do what we can here to factor away the essential logic from the platform specific, but note that getting delayed event services running on any particular platform will require some additional work.

We will try to make as few assumptions about the available timing services of the platform. Here are the constraints on the timing services:

- The is only a single source of timing. That timing source allows us to specify some time value to it and it will respond with some notification when the given time has elapsed.
- It is possible to stop the timing and determine how much time remains before it expires.
- It is not acceptable to execute code periodically that does nothing. This is to account for battery powered devices that cannot afford to wake up and check a timing queue only to find out that there is nothing to do. Activity must be strictly event driven and that implies that we can get positive notification when a time period has elapsed.

4.4 Event Dispatch

Finally, we arrive at the point where we can discuss event dispatching. Up until this time, we have been concerned with generating events, *i.e.* queuing events to be delivered. Now we examine the means by which events are delivered to target instances.

As we discussed above, there are three types of events. Naturally enough, we provide separate functions to dispatch each type of event.

61 ⟨*event dispatch* 61⟩≡ (150 153 156) 62a ▷

```
static void dispatchNormalEvent(MechEcb) ;
static void dispatchPolyEvent(MechEcb) ;
static void dispatchCreationEvent(MechEcb) ;
```

Uses `MechEcb` 24.

We use a simple table to resolve the function selection at run time.

62a ⟨*event dispatch* 61⟩+≡ (150 153 156) ◁61 62b▷

```
static void (* const ecbDispatchFuncs[])(MechEcb) = {
    dispatchNormalEvent,
    dispatchPolyEvent,
    dispatchCreationEvent,
} ;
```

Uses `MechEcb` 24.

The general function to dispatch an event just selects the specific dispatch function based on the event type.

62b ⟨*event dispatch* 61⟩+≡ (150 153 156) ◁62a 65▷

```
static void
mechDispatch(
    MechEcb ecb)
{
    ecbDispatchFuncs[ecb->eventType](ecb) ;
}
```

Uses `MechEcb` 24.

In the sections below, we consider the dispatch of each particular event type.

4.4.1 Normal Event Dispatch

Far and away the most common activity is to dispatch a normal event to a state machine. We call these event types *normal* only to distinguish them from the more complicated polymorphic and creation event types.

Dispatching an event in its simplest terms involves using the current state of the instance and the event number contained in an ECB as indices into the transition matrix. The transition matrix is the same for all instances of a class. The entry in the transition matrix is the new state to which a transition is to be made. There are a few complications such as the need to account for *ignored* and *can't happen* events.

Ignored events cause no transition. Events that are ignored can be thought of as an optimization on the state transition graph. Ignored events can be handled by adding a new state to which the ignored event makes a transition and that new state has all the other outbound transitions that the original state had. Clearly, having the concept of ignored events saves much clutter in the state transition graph.

When the analyst considers a transition to be a logical impossibility, then it is declared as a can't happen event. In the STSA, a can't happen transition is treated as a fatal system error. This is a policy decision of the architecture, so don't be confused that *can't happen* means *shouldn't happen* or *has a low probability of happening*. In this architecture, can't happen means absolutely impossible to happen and if it does happen then there has been a tear in the space / time continuum and the only course available is to give up and declare a fatal error.

The data structure used for normal event dispatch is called an Object Dispatch Block. Each class that has a state machine must supply an ODB. Below we will see how all this ties together. For now, we discuss the data structure and how it is used.

63 ⟨*data types* 10⟩+≡ (149 152 155) ◁23a 73▷
```
typedef struct objectdispatchblock {
    DispatchCount stateCount ;
    DispatchCount eventCount ;
    StateCode const *transitionTable ;
    PtrActionFunction const *actionTable ;
    bool const *finalStates ;
} const *ObjectDispatchBlock ;
```
Uses `PtrActionFunction` 64b.

The state transition matrix is pointed to by the `transitionTable` member. The dimensions of the table are given by the `stateCount` and `eventCount` members.

64a ⟨*base types* 11a⟩+≡ (149 152 155) ◁40 64b▷

```
typedef uint8_t DispatchCount ;
```

Defines:
`uint8_t`, used in chunk 139.

The 8 bit limit for counting is sufficient. Large state machines are definitely a sign of a weak design.

The transition table is in state major order, *i.e.* the current state is used to index conceptual rows and the event number is used to index conceptual columns. The dimensions are captured in the ODB to allow run time bounds checking during event dispatch.

The basic transition algorithm is to use the current state of an instance and the event number of an event as the indices into the transition matrix. The entry in the transition matrix is the new state. Notice the very simple data structures required for Moore state machines.

The new state is used as an index into the `actionTable`. The action table is an array of function pointers to the action associated with each state.

64b ⟨*base types* 11a⟩+≡ (149 152 155) ◁64a 72a▷

```
typedef void ActionFunction(void *const, void *const) ;
typedef ActionFunction *PtrActionFunction ;
```

Defines:
`PtrActionFunction`, used in chunks 63 and 65.

Since Moore state machines associate the action with the state, that code segment is supplied as a function matching the prototype above. The first argument is a pointer to the instance receiving the event. It is `void` typed and state actions are expected to recover the correct type by casting the pointer to the be of the proper instance data structure. The second argument is a pointer to the event parameters. Again, the correct type is recovered in the state action by an appropriate case. Notice that assigning back into event parameters does not make any sense as the parameter values are discarded after the state action completes.

One other feature of the state machine dispatch rules regards final states. A state may be marked as final and if so, then the mechanisms will cause the instance to be destroyed when the state action is completed. The `finalStates` member points to an array, indexed by state number, that specifies if a particular state is indeed a final state. As is frequently the case, the class may have no final states. In this case, `finalState` may be `NULL` to save the storage of the final state booleans.

65 ⟨*event dispatch* 61⟩+≡ (150 153 156) ◁62b 75▷

```
static void
dispatchNormalEvent(
    MechEcb ecb)
{
    MechInstance target = ecb->instOrClass.targetInst ;
    ObjectDispatchBlock db = target->instClass->odb ;

    /*
     * Test for corruption of the current state
     * or event number.
     */
    assert(db->stateCount > target->currentState) ;
    assert(db->eventCount > ecb->eventNumber) ;
    /*
     * Check for the "event-in-flight" error. This occurs
     * when an instance is deleted while there is an event
     * for that instance in the event queue.  For this
     * architecture, such occurrences are considered as
     * run-time detected analysis errors.
     */
    if (target->alloc != ecb->alloc) {
        mechFatalError(mechEventInFlight, ecb->srcInst,
                target, ecb->eventNumber) ;
    }
    /*
     * Fetch the new state from the transition table.
     */
    StateCode newState = *(db->transitionTable +
            target->currentState * db->eventCount +
            ecb->eventNumber) ;
#       ifdef MECH_SM_TRACE
    /*
     * Trace the transition.
     */
    traceNormalEvent(ecb->eventNumber, ecb->srcInst,
            ecb->instOrClass.targetInst, target->currentState, newState) ;
#       endif
    /*
     * Check for a can't happen transition.
     */
    if (newState == MECH_STATECODE_CH) {
        mechFatalError(mechCantHappen, target,
                target->currentState, ecb->eventNumber) ;
    } else if (newState != MECH_STATECODE_IG) {
        /*
         * Check for corrupt transition table.
         */
        assert(newState < db->stateCount) ;
        /*
         * We update the current state to reflect the
         * transition before executing the action for the
         * state.
         */
        target->currentState = newState ;
        /*
         * Invoke the state action if there is one.
         */
        PtrActionFunction action = db->actionTable[newState] ;
        if (action) {
            action(target, &ecb->eventParameters) ;
        }
        /*
         * Check if we have entered a final state. If so,
         * the instance is deleted.
         */
        if (db->finalStates && db->finalStates[newState]) {
            mechInstDestroy(target) ;
        }
    }
    /*
     * Return the ECB to the pool.
     */
    mechEventDelete(ecb) ;
}
```

Uses `MECH_STATECODE_CH` 11c, `MECH_STATECODE_IG` 11c, `MechEcb` 24, `MechInstance` 10, and `PtrActionFunction` 64b.

The processing for dispatching a normal event follows directly from the definitions. After the check for an event-in-flight error, we perform the indexing into the transition matrix. The indexing expression results from the need to treat a linear set of bytes as a two dimensional matrix. We can't type it any differently since we have different sized transition matrices for each different state machine. After obtaining the new state, we must determine if we are actually going to make a transition or if the event is to be ignored or considered a fatal error. Assuming that we are transitioning, then the associated state action is found and executed. Note that empty state actions may be dispensed with and a `NULL` inserted into the action table. After the action, we check if the instance entered a final state.

We are finally in a position to explain the event-in-flight error in detail. The mechanisms detect only one analysis error at run time, the delivery of an event to an instance that has been deleted. Because events are queued, it is possible for an event to be generated for an instance and then while the event is on the queue awaiting to be delivered, the target instance is deleted by some other action code executing. For a single threaded architecture, this is considered an analysis error. In essence this should never happen! The model is responsible for insuring that instance deletion is accomplished only after there are no events awaiting to be delivered. However, it can happen and the mechanisms detect and catch this.

A significant difficulty arises in systems that use distinct memory pools for the instances of each class. If an instance is destroyed and another one created, they may very well end up in exactly the same array slot and therefore have exactly the same instance pointer value. So, a pathological case where an event is generated for an instance, the instance is deleted and then re-created while the event is queued could end up delivering the event to the newly recreated instance. Quite the wrong thing to do.

The strategy used here is to vary the number in the `alloc` field of the instance each time it is allocated. Then a copy of the `alloc` field is placed in the ECB when the event is queued. When dispatched, the values of the `alloc` fields in the two structures must match or else the target instance has been destroyed and re-created in the same memory slot. Of course, the observant reader will have seen that in the case where the target instance is destroyed and recreated 254 times while the event is queued will result in the event being dispatched to the wrong instance. This is considered such a remote possibility as to be of no practical concern.

4.4.2 Polymorphic Event Dispatch

Polymorphic events in their full generality can be complex, but they are based on a simple idea. In fact, there is nothing going on in the dispatch of polymorphic events that could not otherwise be handled in the action code of a state. So, polymorphic events must be considered an optimization, but a very convenient and significant one.

Polymorphic events arise the context of a generalization hierarchy where the subtype classes exhibit different behavior when responding to the same event. Consider the class diagram in figure 4.

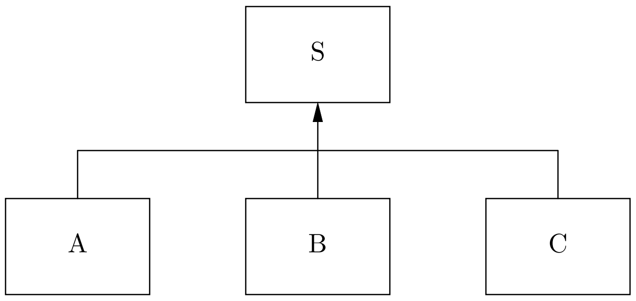

Figure 4: Class Generalization Hierarchy

This figure represents a complete, disjoint partitioning of the instances of S into three subsets, namely the instances of A, B and C. Also suppose that each subtype has a state model and that each of those state models responds to a common set of events, E_s. From the point of view of some arbitrary instance that is generating one of the E_s events to one of the subtypes of S, it is much more convenient to think of generating an event to an instance of S since we know that for each instance of S there is is exactly one instance of either A, B or C. The difficulty arises in that, *a priori*, we do not know which particular subtype instance is related to the instance of S to which we are generating the event. Thus to generate an event to an instance of S that will ultimately be dispatched to one of the subtype instances implies that that we must determine which particular subtype is related to S at the time that the event is dispatched.

Conceptually, determining which subtype is related to a particular instance of a supertype is not difficult. One is required to traverse the relationship from the supertype to each subtype, stopping when the related subtype set is not empty. Beyond the problem of sprinkling a large amount of generically the same code throughout the state actions, such code is very fragile since adding or removing a subtype requires recoding the supertype to subtype queries for each such event generation. Since the system *knows* what the subtypes of any generalization are, polymorphic event dispatch is a factoring into the software architecture of the determination of the currently related subtype and the subsequent mapping of the polymorphic event into an event in the subtype.

Despite the fact that conceptually polymorphic event dispatch is simply the mapping of a polymorphic event into a normal event of the currently related subtype, the full set of rules and implications of polymorphic events is rather daunting.

- Associating polymorphic events with a supertype does *not* imply that the supertype has no state behavior of its own. A supertype may have a state machine and polymorphic events since generating a polymorphic event to a supertype does not result in any behavior in the supertype.

- A given supertype class may be the supertype of multiple generalization hierarchies. In this case, generating an event to an instance of such a supertype will cause an event to be generated down all hierarchies for which the class is a supertype. In this way, a single event generated may turn into multiple events being dispatched.
- The state machine for a subtype class may respond to normal events that are not part of the polymorphic event set associated with the generalization hierarchy. Such events may be generated directly to instances of the subtype class or they may be self generated.
- A class that is a subtype may also be a supertype of another generalization hierarchy, *i.e.* the depth of a generalization hierarchy may be greater than one. Such a mid-level class may have new polymorphic events associated with the generalization hierarchy for which it is the supertype. Also a mid-level class may delegate polymorphic events associated with its subtype role to any hierarchy for which it serves as a supertype.
- All leaf classes, *i.e.* subtype classes which are *not* the supertype of another hierarchy, must consume as normal events all polymorphic events delegated down the hierarchy to them. It is sufficient to ignore or deem an event as can't happen, but polymorphic events must be ultimately mapped to normal events.

To illustrate some of these rules, consider the more complicated generalization hierarchy shown in figure 5. Assume that the polymorphic events, $E1_{S0}$ and $E2_{S0}$ are associated with class S0. Generating $E1_{S0}$ to an instance of S0 will result in an event being dispatch down both the R1 and R4 hierarchies. Class S11, for example, may consume $E1_{S0}$ as a normal event or may delegate it to be dispatched polymorphically to its subtypes, S21 and S22. Further, either classes S11 or S12 may associate additional polymorphic events to the R2 and R3 hierarchies, respectively. Eventually and regardless of where in the hierarchy they are introduced, all the polymorphic events must map to normal events in the leaves or be consumed as normal events by the instances of the intermediate classes. Also note that S0 may or may not have its own state machine. The fact that it has polymorphic events does *not* necessarily imply that it has an associated state machine. It does imply that at least the leaf classes of the hierarchy do have an associated state machine.

Generalization Relationship Storage

There are two fundamental steps in dispatching a polymorphic event:

1. Determining which subtype instance is currently related to a supertype instance.
2. Mapping the polymorphic event encoding in the supertype to an event encoding in the subtype.

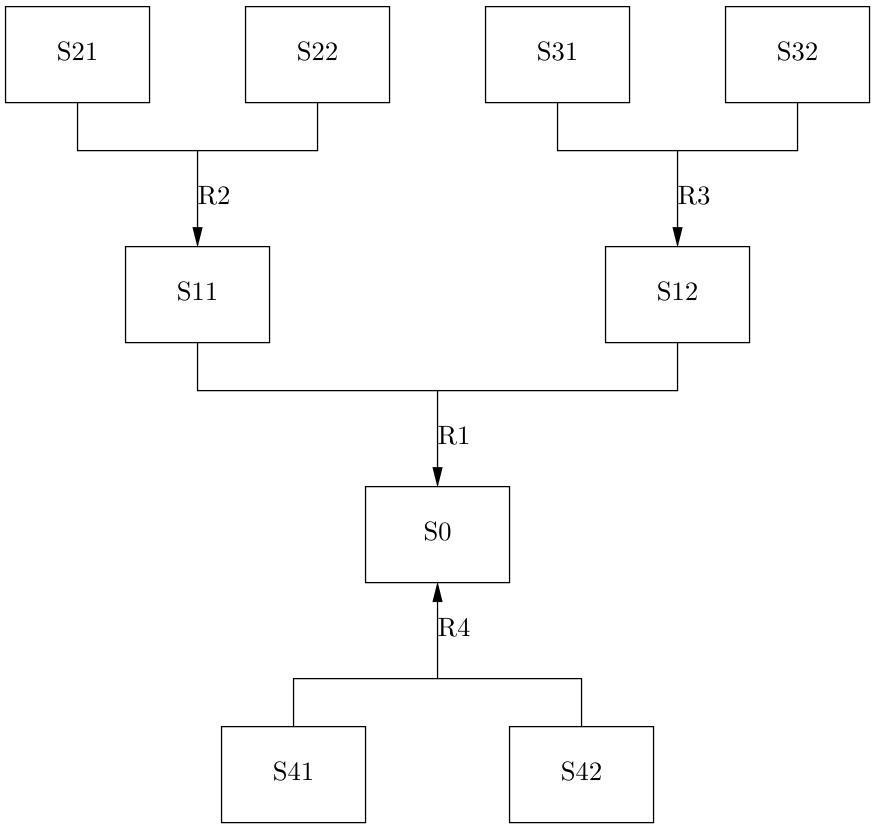

Figure 5: Complex Class Generalization Hierarchy

In order to accomplish the first step, the mechanisms have know how the generalization relationship is stored in the instances. The mechanisms support two different schemes of storing the generalization relationship, either as a pointer reference or as a union.

Both relationship storage techniques have their uses and we will not discuss the pros and cons of one choice over another here. If the generalization relationship is stored as a reference, then the supertype instances must must contain an instance pointer to a subtype. If the generalization relationship is stored as a union, then the supertype instances are presumed to have a member that is a union of the data types of all the subtypes of the generalization hierarchy. This distinction is an attribute of the class and is encoded in an enumerated data type.

72a ⟨*base types* 11a⟩+≡ (149 152 155) ◁64b 72b▷
```
typedef enum {
    PolyReference,
    PolyUnion
} PolyStorageType ;
```
Defines:
`PolyStorageType`, used in chunk 73.

It is also necessary to store in the supertype instance some encoding of the subtype to which it is currently related. This is just a simple zero based, sequential number that is associated with the subtypes of the hierarchy.

72b ⟨*base types* 11a⟩+≡ (149 152 155) ◁72a 72c▷
```
typedef uint8_t SubtypeCode ;
```
Defines:
`uint8_t`, used in chunk 139.

As we will see, if we can locate where in the supertype instance structure the subtype encoding and the subtype reference or union are located, then we can determine the type of the subtype instance to which a particular supertype instance is related. To do that we assume that the there is a data type that can hold the byte offset from the beginning of the supertype instance structure to the required information. As you can probably imagine, this will be a tricky piece of code since it must pick out information from an arbitrary data structure in a generic fashion.

72c ⟨*base types* 11a⟩+≡ (149 152 155) ◁72b 88▷
```
typedef unsigned short int AttributeOffset ;
```
Defines:
`AttributeOffset`, used in chunk 73.

Polymorphic Event Mapping

We now turn our attention to the actual mapping of polymorphic events. The mapping is analogous to the mapping of current state and event to a new state for normal event dispatch. For polymorphic events, the mapping is from subtype code of the currently related subtype and polymorphic event number to a new event. The data structure required for this is given by:

73 ⟨*data types* 10⟩+≡ (149 152 155) ◁63 74a▷

```
typedef struct hierarchydispatch {
    PolyStorageType refStorage ;
    AttributeOffset subCodeOffset ;
    AttributeOffset subInstOffset ;
    DispatchCount subtypeCount ;
    struct polyeventmap const *eventMap ;
} const *HierarchyDispatch ;
```

Uses `AttributeOffset` 72c and `PolyStorageType` 72a.

`refStorage` The `refStorage` member determines if the generalization relationship is stored in reference form or in union form.

`subCodeOffset` The `subCodeOffset` member holds the byte offset from the beginning of the instance structure where the encoding for the currently related subtype is held.

`subInstOffset` The `subInstOffset` member holds the byte offset from the beginning of the instance structure where either a pointer to a subtype is stored or the union for the subtypes is located.

`subtypeCount` The `subtypeCount` member holds the number of distinct subtypes that there are for this generalization relationship. This value is used for run-time checks.

`eventMap` The `eventMap` member is a pointer to the mapping of polymorphic events for the hierarchy. This mapping is indexed by major order by subtype code and in minor order by polymorphic event number.

A key realization here is that as we are mapping down a hierarchy a given polymorphic event may be mapped into a normal event where it will be consumed by the state machine of the class or it may be delegated further down the hierarchy. To be further delegated implies that a polymorphic event will be mapped into yet another polymorphic event to be further mapped in a subsequent dispatch. Thus the entries in the `eventMap` matrix contain both a new event number and an indication of the type of the new event.

74a ⟨*data types* 10⟩+≡ (149 152 155) ◁73 74b▷
```
typedef struct polyeventmap {
    EventCode event ;
    MechEventType eventType ;
} const *PolyEventMap ;
```

Uses `MechEventType` 21.

Now we can tie it all together. A supertype class that has associated polymorphic events must supply a Polymorphic Dispatch Block (PDB) to direct the mechanisms as to how to perform the mapping of polymorphic events to normal events.

74b ⟨*data types* 10⟩+≡ (149 152 155) ◁74a 79b▷
```
typedef struct polydispatchblock {
    DispatchCount eventCount ;
    DispatchCount hierCount ;
    HierarchyDispatch hierarchy ;
} const *PolyDispatchBlock ;
```

`eventCount` The `eventCount` member holds the number of polymorphic events associated with the supertype class. Like normal events, polymorphic events are encode as zero based sequential integers so they may be used as array indices in the mapping process.

`hierCount` The `hierCount` member holds the number of generalization hierarchies that extend from the supertype class.

`hierarchy` The `hierarchy` member holds a pointer to an array of Hierarchy Dispatch Blocks. The array contains `hierCount` elements.

Now we can give the code for polymorphic event dispatch.

75 ⟨*event dispatch* 61⟩+≡ (150 153 156) ◁65 78▷

```
static void
dispatchPolyEvent(
    MechEcb ecb)
{
    PolyDispatchBlock pdb = ecb->instOrClass.targetInst->instClass->pdb ;

    assert(pdb != NULL) ;
    assert(ecb->eventNumber < pdb->eventCount) ;
    assert(pdb->hierCount > 0) ;
    /*
     * Each generalization hierarchy that originates at the
     * supertype has an event generated down that
     * hierarchy to one of the subtypes.
     */
    HierarchyDispatch hd = pdb->hierarchy ;
    for (unsigned hnum = 0 ; hnum < pdb->hierCount ; ++hnum) {
        /*
         * The most common case is to dispatch along a
         * single hierarchy.  In any case, we can modify in
         * place the input ECB on the last dispatched event.
         */
        MechEcb newEcb ;
        if (hnum == pdb->hierCount - 1) {
            newEcb = ecb ;
        } else {
            newEcb = mechEventAlloc() ;
            /*
             * We set the source as the original sender.
             */
            newEcb->srcInst = ecb->srcInst ;
            /*
             * Copy event parameters.
             */
            newEcb->eventParameters = ecb->eventParameters ;
        }
        SubtypeCode type =
                *(SubtypeCode *)((char *)ecb->instOrClass.targetInst +
                hd->subCodeOffset) ;

        assert(type < hd->subtypeCount) ;
        PolyEventMap pem = hd->eventMap +
                (type * pdb->eventCount + ecb->eventNumber) ;
#           ifdef MECH_SM_TRACE
        /*
         * Trace the transition.
         */
        tracePolyEvent(ecb->eventNumber, ecb->srcInst,
                ecb->instOrClass.targetInst, type, hnum,
                pem->event, pem->eventType) ;
#           endif /* MECH_SM_TRACE */

        newEcb->eventNumber = pem->event ;
        newEcb->eventType = pem->eventType ;

        void *subTypeRef =
                (char *)ecb->instOrClass.targetInst + hd->subInstOffset ;
        newEcb->instOrClass.targetInst = hd->refStorage == PolyReference ?
            /*
             * When the generalization is implemented via a
             * pointer, we need an extra level of
             * indirection to fetch the address of the
             * subtype.
             */
            *(MechInstance *)subTypeRef :
            /*
             * When the generalization is implemented by a
             * union, we need only point to the address of
             * the subtype as it is contained in the
             * supertype.
             */
            (MechInstance)subTypeRef ;

        if (newEcb->eventType == NormalEvent) {
            newEcb->alloc = newEcb->instOrClass.targetInst->alloc ;
            assert(newEcb->alloc != 0) ;
        }

        mechDispatch(newEcb) ;
        ++hd ;
    }
```

```
  }
```
Uses `MechEcb` 24 and `MechInstance` 10.

The code loops through all of the hierarchies for which the `targetInst` is a supertype. The vast majority of the time there is only one hierarchy. The strategy used here is to reuse the ECB that was carrying the polymorphic event as the ECB for the last hierarchy. Otherwise, if there are multiple hierarchies additional ECB will need to be allocated. Any newly allocated ECB must carry the same source instance and event parameter information.

The core of the algorithm is to fetch the subtype code from the instance and use that as the row index into the polymorphic event map for the hierarchy. The event number is then used as the column index to find the mapping entry. That mapping entry contains a new event number and event type. The new target of the event is then the currently related subtype instance. As discussed above, this may be a stored as a pointer or may be a union member of the supertype instance structure. In the first case, we fetch the pointer from its location in the supertype structure. For the union case, the location in the supertype structure is the beginning of the union, *i.e.* we down cast to the subtype member. Should the mapped event resolve to be a normal event, then we must fill in the `alloc` field to enable the event in flight detection. Finally the newly minted ECB is recursively dispatched and the next hierarchy is considered. Recursively dispatching the event preserves the order of delivery of the events.

As you can see here, the code to implement the dispatch is not particularly complicated. The complications arise in supplying the data required for the polymorphic event mapping. Again tooling is very helpful here as all the indices and offsets must be correctly supplied.

4.4.3 Creation Event Dispatch

Fortunately, creation events are much simpler than polymorphic events. Creation event dispatch is the simple combination of instance allocation and normal event dispatch. No additional data structures are required. The code below shows how it is done.

78 ⟨*event dispatch* 61⟩+≡ (150 153 156) ◁75

```
static void
dispatchCreationEvent(
    MechEcb ecb)
{
    /*
     * For creation events we must allocate an instance,
     * set the state to be the creation state (by
     * convention the creation state is 0).
     */
    MechInstance inst = mechInstCreate(ecb->instOrClass.targetClass,
            MECH_DISPATCH_CREATION_STATE) ;
#           ifdef MECH_SM_TRACE
    /*
     * Trace the transition.
     */
```

```
        traceCreationEvent(ecb->eventNumber, ecb->srcInst,
                inst, ecb->instOrClass.targetClass) ;
    #           endif
        /*
         * Modify the event structure in place and dispatch it.
         */
        ecb->instOrClass.targetInst = inst ;
        ecb->eventType = NormalEvent ;
        ecb->alloc = ecb->instOrClass.targetInst->alloc ;
        assert(ecb->alloc != 0) ;

        dispatchNormalEvent(ecb) ;
    }
```

Uses `MECH_DISPATCH_CREATION_STATE` 79a, `MechEcb` 24, and `MechInstance` 10.

The dispatch reuses the ECB that contains the creation event. A new type is assigned and the target instance is created. The remaining fields are filled in and the ECB is recursively dispatched.

The only concept that needs additional explanation is that of a creation pseudo-state. Classes that have creation events have an initial pseudo-state from which the instance can transition when the creation event is dispatched. By convention, that state is numbered zero for convenience. The new created instance is placed in this state and the event can then be dispatched in the same manner as any other normal event.

79a ⟨*constants* 11c⟩+≡ (149 152 155) ◁28a 96b▷

```
    #define MECH_DISPATCH_CREATION_STATE    0
```

Defines:
`MECH_DISPATCH_CREATION_STATE`, used in chunk 78.

4.4.4 Class Data

By now it should be evident that the mechanisms are completely data driven. All the behavior of data management and execution sequencing is completely determined by the values contained in the data structures supplied to the various functions of the mechanisms. This is different from some software architecture mechanisms that use *ad hoc* generated code from a model compiler to implement some capabilities.

Since the behavior of all instances of a given class is the same, each class that requires data management or has execution behavior must supply a data structure that contains all the class invariant information.

79b ⟨*data types* 10⟩+≡ (149 152 155) ◁74b 89c▷

```
    typedef struct mechclass {
        InstAllocBlock iab ;
        ObjectDispatchBlock odb ;
        PolyDispatchBlock pdb ;
    } const *MechClass ;
```

Uses `InstAllocBlock` 13.

That information is a pointer to an instance allocation block, a pointer to an object dispatch block and a pointer to a polymorphic dispatch block. Any class that does not need a particular facility may supply `NULL` for that block. Presumably not all the members of the `MechClass` are `NULL` since such classes are pure data that has a constant population given by its initial instances and therefore have no need to define the `MechClass` structure.

4.5 Event Dispatch Tracing

Debugging event driven, callback, state machine based applications can be rather more complicated than conventional, synchronously designed code. One instance generates an event to another instance and it can be hard to determine the exact sequence of execution by simply examining the source code. Indeed, part of the intent here is to factor away from the application the details of sequencing execution. Setting a breakpoint in the action of a state is easy enough, the difficulties arise when trying to determine where to set a breakpoint to catch the results of the next event dispatch. Given that many events will be flying around a given program, it is very useful to be able to extract the set of event dispatches in chronological order.

To help in debugging, the mechanisms can be conditionally compiled to support tracing the event dispatch. After the code is properly compiled, a pointer to a trace callback function may be registered and then each event dispatched will result in the function being called with the information about the dispatch.

It should be noted that tracing information is very primitive. Only numbers are involved as we do not put strings into the target executable. This means that there is substantial effort required to back translate the numbers into strings that are meaningful to a human. Again, programs such as **pycca** can be very helpful in this effort.

80 ⟨*conditional compilation* 80⟩≡ (149 152 155) 89b▷
```
* If the symbol MECH_SM_TRACE is defined to the preprocessor,
* then code supporting state machine dispatch tracing is
* included in the object file.
```

4.5.1 Trace Information

Since there are three types of events, there are three distinct sets of information generated when an event is dispatched. There is common information for all events and three sets of information specific to each event type.

81 ⟨*trace data types* 81⟩≡ (149 152 155) 82a ▷

```
typedef struct mechtraceinfo {
    MechEventType eventType ;
    EventCode eventNumber ;
    void *srcInst ;
    void *dstInst ;
    union {
        struct normaltrace {
            StateCode currState ;
            StateCode newState ;
        } normalTrace ;
        struct polytrace {
            SubtypeCode subcode ;
            DispatchCount hierarchy ;
            EventCode mappedNumber ;
            MechEventType mappedType ;
        } polyTrace ;
        struct createtrace {
            void const *dstClass ;
        } creationTrace ;
    } info ;
} *MechTraceInfo ;
```

Defines:
`MechTraceInfo`, used in chunk 82a.
Uses `MechEventType` 21.

- Data common to all event dispatch traces.

  `eventType` The type of the event that was dispatched.

  eventNumber The number of the event that was dispatched.

  srcInst A pointer to the instance that was the source of the dispatched event.

  dstInst A pointer to the instance that was the target of the dispatched event.

- Data for normal event dispatch traces.

  currState The current state of the instance before the event dispatch.

  newState The new state entered as a result of the transition.

- Data for polymorphic event dispatch traces.

  subcode The subtype code of the currently related instance.

  hierarchy The number of the hierarchy down which the event was dispatched.

  mappedNumber The new event number to which the polymorphic event mapped.

  mappedType The new event type corresponding to `mappedNumber`.

- Data for creation event dispatch traces.

  dstClass A pointer to the class structure for the target of the creation event.

4.5.2 Access to Trace Information

Event tracing information is passed out of the mechanisms by having the application register a callback function. That function takes a pointer to the trace information as its argument.

82a ⟨*trace data types* 81⟩+≡ (149 152 155) ◁81

```
typedef void (*MechTraceCallback)(MechTraceInfo) ;
```

Defines:
`MechTraceCallback`, used in chunks 82b and 83.
Uses `MechTraceInfo` 81.

The function is registered with the mechanisms by invoking:

82b ⟨*trace external functions* 82b⟩≡ (149 152 155)

```
extern MechTraceCallback mechRegisterTrace(MechTraceCallback) ;
```

Uses `MechTraceCallback` 82a.

The trace callback function is supplied as the argument and the previous value of the callback is returned. Tracing can be turned off by invoking `mechRegisterTrace` with `NULL`.

83 ⟨*event tracing* 83⟩≡ (150 153 156) 84▷

```
static MechTraceCallback traceCallback ;

MechTraceCallback
mechRegisterTrace(
    MechTraceCallback cb)
{
    MechTraceCallback oldcb = traceCallback ;
    traceCallback = cb ;
    return oldcb ;
}
```

Defines:
`traceCallback`, used in chunks 84–86.
Uses `MechTraceCallback` 82a.

The implementation of registering a callback is simply to record the function pointer in a variable.

For each type of event dispatch, the mechanisms call a specific function to determine if tracing is enabled and to marshal the trace information into the proper data structure.

84 ⟨*event tracing* 83⟩+≡ (150 153 156) ◁83 85▷

```
static inline
void
traceNormalEvent(
    EventCode event,
    MechInstance source,
    MechInstance target,
    StateCode currentState,
    StateCode newState)
{
    if (traceCallback) {
        struct mechtraceinfo trace ;

        trace.eventType = NormalEvent ;
        trace.eventNumber = event ;
        trace.srcInst = source ;
        trace.dstInst = target ;
        trace.info.normalTrace.currState = currentState ;
        trace.info.normalTrace.newState = newState ;

        traceCallback(&trace) ;
    }
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechInstance` 10 and `traceCallback` 83.

85 ⟨*event tracing* 83⟩+≡ (150 153 156) ◁84 86▷

```
static inline
void
tracePolyEvent(
    EventCode event,
    MechInstance source,
    MechInstance target,
    SubtypeCode subtype,
    DispatchCount hierarchy,
    EventCode newEvent,
    MechEventType newEventType)
{
    if (traceCallback) {
        struct mechtraceinfo trace ;

        trace.eventType = PolymorphicEvent ;
        trace.eventNumber = event ;
        trace.srcInst = source ;
        trace.dstInst = target ;
        trace.info.polyTrace.subcode = subtype ;
        trace.info.polyTrace.hierarchy = hierarchy ;
        trace.info.polyTrace.mappedNumber = newEvent ;
        trace.info.polyTrace.mappedType = newEventType ;

        traceCallback(&trace) ;
    }
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechEventType` 21, `MechInstance` 10, and `traceCallback` 83.

86 ⟨*event tracing* 83⟩+≡ (150 153 156) ◁85
```
  static inline
  void
  traceCreationEvent(
      EventCode event,
      MechInstance source,
      MechInstance target,
      MechClass class)
  {
      if (traceCallback) {
          struct mechtraceinfo trace ;

          trace.eventType = CreationEvent ;
          trace.eventNumber = event ;
          trace.srcInst = source ;
          trace.dstInst = target ;
          trace.info.creationTrace.dstClass = class ;

          traceCallback(&trace) ;
      }
  }
```
Defines:
`inline`, used in chunk 101b.
Uses `MechInstance` 10 and `traceCallback` 83.

4.5.3 Tracing Strategies

Clearly, tracing can generate data at a rather high rate and can be rather intrusive upon the execution of the system. Several strategies may be used to deal with the trace data. If possible, all the trace data can be dumped in a raw form out a communications interface and let some other program decode and display it. That may still be too intrusive and sometimes it is best to filter the trace data based on the target instance pointer value. In this way you may trace the event dispatches on only a subset of instances. Several different filtering schemes, such as source instance or classes, can be envisioned.

Another possibility is to store trace information in a memory area in some sort of circular queue arrangement. Then it is possible for the application to start and stop such tracing and achieve "logic analyzer" type triggering functionality. The trace information can then be extracted from memory and analyzed.

You will also note that the trace information has no timing data associated with it. This type of data is so system specific that it is left to the tracing callback to supply. If you have a free running cycle counter in your system, this can be a good indicator of relative time and the trace callback function can add this to the data set supplied by the mechanisms. Your system may also have source of clocked timing data that can also be used as a timing reference. In either case, augmenting the trace data with some sort of relative time information is very valuable.

Tracing can also be used as a framework for testing. If a domain is built to run in a testing framework where tracing is enabled, then recording all the trace information allows one to determine the amount of *transition* coverage a test set causes. The goal is to develop test sets that drive the domain by invoking the domain interface functions with appropriate data so that all state transitions are taken. Tracing allows the recording of what transitions a given thread of control causes. Since in most well designed state machines, state action code is small and does not contain complicated or intricate internal program flow, causing all state actions to be executed is often close to complete statement coverage. As an added benefit, state machines can be considered as directed graphs. A depth first traversal of a directed graph can be be used to determine a spanning tree for the graph. Traversing a spanning tree for a graph insures that all nodes in the graph are visited and the event sequence given by the spanning tree can guide the generation of test set data and can help to minimize the number of test cases required to ensure adequate coverage.

5 Error Handling

Until now we have glossed over the subject of how to handle errors in the software mechanisms. In XUML, the domains assume a perfect architecture in the sense that no formal mechanism is provided to signal architectural errors back to the application domains. This makes sense because the application models are meant to be implementation independent and able to be run on a variety of underlying mechanisms. However, an error policy, in much the same terms as data and execution policies, must be put into place. The details of the error handling policy will vary between software architectures, so it is important to state them clearly. For the STSA, the following principles guide error handling.

- To that extent possible, the mechanisms operations should not report errors back to the application. For implementation languages that do not support exception handling, the usual technique of returning error codes is not very effective. Either by accident or sloth, many error codes are not checked. Even then the error code is checked, there is little recovery recourse for the application. For example, it does little good to know that we are unable to generate an event because we do not have sufficient ECB resources when there is nothing a state action can do to free up the required resources.
- Errors that result from exhausted resources or analysis errors detected at run time are *fatal*. Exactly how fatal errors are acted upon is platform dependent and may result in terminating a program or completely resetting the system. Regardless of the consequence of a fatal error, the assumption is that the program can no longer continue to run.

With these principles in mind, we define a set of error conditions that are detected by the mechanism. All these conditions are fatal and are handled by invoking a fatal error handler.

88 ⟨*base types* 11a⟩+≡ (149 152 155) ◁72c

```
typedef enum {
    mechCantHappen = 1,
    mechEventInFlight,
    mechNoECB,
    mechNoInstSlot,
    mechSyncOverflow,
    #ifdef __unix__
    mechTimerOpFailed,
    mechSignalOpFailed,
    mechSelectWaitFailed,
    #endif /* __unix__ */
} MechErrorCode ;
```

Defines:
`MechErrorCode`, used in chunks 89–91 and 150.

We will need a string representation of the error codes to make human readable messages.

89a ⟨*error handling* 89a⟩≡ (150 153 156) 90a ▷

```
static char const * const errMsgs[] = {
    "no error",     /* place holder */
    "can't happen transition: %p: %u - %u -> CH\n",
    "event in flight error: %p -> %p %u\n",
    "no available Event Control Blocks\n",
    "no available instance slots: %p\n",
    "synchronization queue overflow\n",
    #ifdef __unix__
    "interval timer operation failed: %s\n",
    "signal operation failed: %s\n",
    "blocking on pselect() failed: %s\n",
    #endif /* __unix__ */
} ;
```

Defines:
`errMsgs`, used in chunks 90a and 91a.

Everywhere else the mechanisms operations have been crafted to avoid error possibilities. For example, as discussed in delayed event generation, we interpret the attempt to generate a duplicate delayed event as wishing to cancel the existing one and instantiate a new one. This semantic interpretation avoids generating an error and avoids all the additional code require in state actions that generate delayed events.

Exactly how fatal errors are handled will depend upon the specifics of how the platform handles errors. Here we provide a default that simply prints a message and exits. For bare metal systems, printing may not be an option and so is included conditionally.

89b ⟨*conditional compilation* 80⟩+≡ (149 152 155) ◁ 80 101a ▷

```
 * If the symbol MECH_NINCL_STDIO is defined to the preprocessor,
 * then code supporting printing of fatal error messages will be
 * removed from the object file.
```

We define an interface of a fatal error handler.

89c ⟨*data types* 10⟩+≡ (149 152 155) ◁ 79b 96c ▷

```
typedef void (*MechFatalErrHandler)(MechErrorCode, char const *, va_list) ;
```

Defines:
`MechFatalErrHandler`, used in chunks 90 and 91b.
Uses `MechErrorCode` 88.

The interface is patterned after `vprintf`, giving a format string and a pointer to a variable length argument list.

The system provides a default fatal error handler. Assuming that standard I/O is included, we print a message to the standard error.

90a ⟨*error handling* 89a⟩+≡ (150 153 156) ◁89a 90b▷

```
static void
MechDefaultFatalErrorHandler(
    MechErrorCode errNum,
    char const *fmt,
    va_list ap)
{
#   ifndef MECH_NINCL_STDIO
    vfprintf(stderr, errMsgs[errNum], ap) ;
#   endif /* MECH_NINCL_STDIO */
}
```

Uses `errMsgs` 89a and `MechErrorCode` 88.

A pointer to the fatal error handler is initialized with the default one.

90b ⟨*error handling* 89a⟩+≡ (150 153 156) ◁90a 90c▷

```
static MechFatalErrHandler errHandler = MechDefaultFatalErrorHandler ;
```

Defines:
`errHandler`, used in chunks 90c and 91a.
Uses `MechFatalErrHandler` 89c.

Because fatal error handling is usually so platform specific and because of the need to test fatal error paths, we provide the ability to delegate the consequence of the fatal error.

90c ⟨*error handling* 89a⟩+≡ (150 153 156) ◁90b 91a▷

```
MechFatalErrHandler
mechSetFatalErrHandler(
    MechFatalErrHandler newHandler)
{
    MechFatalErrHandler prevHandler = errHandler ;
    if (newHandler) {
        errHandler = newHandler ;
    }
    return prevHandler ;
}
```

Uses `errHandler` 90b and `MechFatalErrHandler` 89c.

The mechanisms internally call `mechFatalError`. This function is presented next.

91a ⟨*error handling* 89a⟩+≡ (150 153 156) ◁90c

```
static void
mechFatalError(
    MechErrorCode errNum,
    ...)
{
    va_list ap ;
    /*
     * All hope is lost here. Make sure we don't
     * execute any asynchronous code.
     */
    beginCriticalSection() ;

    assert(errHandler != NULL) ;
    assert(errNum < (sizeof(errMsgs) / sizeof(errMsgs[0]))) ;

    va_start(ap, errNum) ;
    errHandler(errNum, errMsgs[errNum], ap) ;
    /*
     *  If the handler does return, we insist that all errors
     *  are fatal. So we exit() unless we are testing.
     */
#   ifndef MECH_TEST
    exit(errNum) ;
#   endif /* MECH_TEST */
}
```

Uses `errHandler` 90b, `errMsgs` 89a, and `MechErrorCode` 88.

As we see in the code, we insist that there is an error handler. There is always the default one and a different handler can be specified if necessary. Finally, exit is called and the exit code is devised from the mechanisms error code.

There is a function to call to supply an error handler. It returns the previous value of the error handler.

91b ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁58 93a▷

```
extern MechFatalErrHandler
        mechSetFatalErrHandler(MechFatalErrHandler) ;
```

Uses `MechFatalErrHandler` 89c.

The implementation of the function to set the error handler is trivial, however, note that the function may be called with a NULL handler value to simply obtain a pointer to the current error handling function.

Avoiding Fatalities

In this error handling strategy, every mechanism detected error is fatal. Although the details of the processing for fatal errors can be delegated, in most systems of the class we consider here, fatal errors usually result in a system reset. Under the vast majority of circumstances, that is the desired behavior. However, there are some particular circumstances where causing a fatal error is not the desired behavior.

Consider the case where some external stimulus results in an event being generated. If the stimulus occurs more frequently than events can be processed, then the mechanisms will run out of event control blocks causing a fatal error. As an example, consider the arrival of a communications packet. If somewhere during the processing of the packet an event is generated, then if packets arrive too fast a fatal error can be generated. In effect it would provide a means for an external stimulus to cause the system to crash. Certainly for the case of a communications packet, the preferred behavior would be to drop the packet and let higher level protocol deal with the necessary retries, etc. It is then necessary to be able to determine if generating an event would be successful.

In this section we describe functions that can be used to avoid mechanisms requests that would exhaust an underlying resource and therefore cause a fatal system error. It should be emphasized that these functions are *not* intended for ordinary or casual use. Under the vast majority of circumstances, such as when one state machine generates an event to another state machine, event generation and other such activities should continue to assume that there are no resources that can be consumed. System analysis and testing should then determine the appropriate sizing for the various resource pools. The capability described in this section is to handle unusual and extraordinary circumstances where hardware failure or failure to abide by communications protocols could force the system into a fatal error situation.

Note also that the alternative provided here causes the external stimulus that would cause the fatal error condition effectively to be ignored. For some system requirements that is not an acceptable solution. For example, consider a digital input line that is used to generate an interrupt and that interrupt signals an external condition monitored by hardware, say the maximum extent of motion of physical robot arm. If this interrupt arrives at a very fast rate, one might conclude the hardware has failed. Ignoring the interrupt might do little other than mask a problem that should cause a fatal error condition and potentially reset the system. The conclusion is that providing a means of avoiding fatal error conditions is not intented to serve as an overall error handling policy. Careful analysis and consideration is still required. If an interrupt arrives faster than expected and, because of what the interrupt represents, it cannot be ignored, that fact and the response to it must be deduced by the interrupt service code (e.g. by determining the interrupt frequency) and actions appropriate to the system must be taken. It can be a difficult problem to solve and the functions provided here are too generic to be of much help.

There are three internal mechanisms resources that can be exhausted.

- Event control blocks are used for generating and dispatching state machine events.
- Class instances can be dynamically created and each class has its own instance pool.
- The foreground / background synchronization queue has a fixed number of slots and excessive synchronization requests from interrupt service routines can fill the queue.

To determine if there is an event control block available the `mechEventAvail` function may be invoked.

93a ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁91b 93c▷

```
extern bool mechEventAvail(void) ;
```

If `mechEventAvail` returns true, then an immediate invocation of `mechEventNew`, `mechPolyEventNew` or `mechCreationEventNew` will not cause a fatal system error.

93b ⟨*event generate* 31a⟩+≡ (150 153 156) ◁36

```
bool
mechEventAvail(void)
{
    return !eventQueueEmpty(&freeEventQueue) ;
}
```

Uses `freeEventQueue` 28c.

The implementation is simple enough since available event control blocks are held on a free queue.

To determine if there is an available instance slot in the storage pool for a class, `mechInstAvail` may be invoked.

93c ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁93a 95▷

```
extern bool
mechInstAvail(
    MechClass instClass) ;
```

If `mechInstAvail` returns true, then an immediate invocation of `mechInstCreate` will not cause a fatal error situation. *N.B.* this does not apply to asynchronously created instances done via creation event. Since a creation event is queued, there is no way to determine *a priori* that an instance slot will be available at the time the event is dispatched.

94 ⟨*instance functions* 14⟩+≡ (150 153 156) ◁19

```
bool mechInstAvail(
    MechClass instClass)
{
    return mechInstFindSlot(instClass) != NULL ;
}
```

The implementation is also simple, calling upon the common code used to find an instance slot in a class storage pool.

Finally, we will defer discussing overflow of the sync queue until the next section below on asynchronous processing.

6 Asynchronous Execution

When the main loop was discussed (p. 5), we made a brief mention of asynchronous execution in the context of executing synchronization functions. It is now time to discuss how the mechanisms deal with asynchronous execution. First we discuss some background and then present the means used by the mechanisms to synchronize the two execution contexts.

Until now, the discussion of data management and execution sequencing have assumed that we have a single execution context. However, all modern computer architectures support the concept of an interrupt. An interrupt signaled by some external hardware causes the processor to stop executing and transfer control to a different section of code so that immediate action can be taken on the cause of the interrupt. After the external condition is handled, execution can resume where it was interrupted. The need for asynchronous execution was recognized early in computer architecture design as interaction with the external environment is much of what makes a computer a useful machine.

The exact details of how this happens is different for every computer architecture. Some computers offer very sophisticated schemes that include arbitrating the priority of multiple competing interrupt sources. Most offer only a single priority of interrupt processing or place the burden of prioritization on the programmer or external hardware. The STSA is intended for highly embedded systems and, in such systems, achieving the low execution overhead is of great value. So the model of asynchronous execution used by the mechanisms mirrors that which is provided for directly by the hardware architecture. Techniques for having multiple, scheduled execution contexts are very well known. None of them is used here. Here we are only interested in a very simple model of asynchronous execution that closely mirrors what is provided by the computing hardware. Amazingly, this is much less restrictive than might be first imagined.

6.1 Simple Interrupt Priority

For processor architectures that only support a simple interrupt scheme (and this includes the POSIX environment where signals serve the purpose of interrupts), we can visualize the interrupt as shown in the figure below.

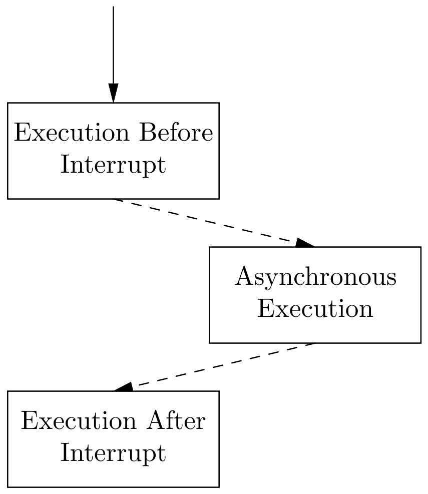

Figure 6: Simple Asynchronous Execution

Figure 6 shows the simple idea that the single execution context is interrupted in order to run a single interrupt service. Upon completion, execution resumes where it left off. Nesting interrupts is not forbidden, but is definitely not encouraged. No restriction is put on what may happen at interrupt service. This is usually a great source of error in many systems. Because the execution is asynchronous to domain model execution, any access to the domain data structures from interrupt service *must not* be attempted. It is ultimately unsafe and results in horribly difficult to detect timing windows being generated wherein things will mysteriously and irreproducibly fail.

However, it is usually necessary for an interrupt to communicate back to the domain models. Usually, the interrupt signals some change of condition in the environment that has been detected by the hardware. The interrupt may be able to do everything require to handle the situation, but more often than not, additional computation is required to resolve what has happened. So the mechanisms provide a way for an interrupt to request synchronization with the running background. This is accomplished by having the interrupt service code perform a synchronization request.

95 ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁93c 96a▷

```
extern SyncParamRef mechSyncRequest(SyncFunc) ;
```

Uses `SyncFunc` 97b and `SyncParamRef` 97a.

That request is in the form of a function with optional parameters that is to be executed at the first *safe* opportunity. As we saw before, the synchronization functions are executed in between state actions to insure that the domain data are in a coherent state.

There is a secondary form to request a background synchronization for those cases where we do not wish to create a fatal system error condition if the sync queue overflows.

96a ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁95 100a▷

```
extern SyncParamRef mechTrySyncRequest(SyncFunc) ;
```

Uses `SyncFunc` 97b and `SyncParamRef` 97a.

This function will return a `NULL` value if the attempt to queue the sync request failed.

The mechanisms provide a sync queue where interrupts may place their requests for background synchronization execution. We do this with a simple queue that is implemented in an array. Like all resources in the mechanisms, the storage required for the queue is fixed at compile time.

96b ⟨*constants* 11c⟩+≡ (149 152 155) ◁79a

```
#ifndef MECH_SYNCQUEUESIZE
#   define MECH_SYNCQUEUESIZE 10
#endif /* MECH_SYNCQUEUESIZE */
```

The number of queue entries can be sized appropriately for the system. Generally, the number of queue slots must be sized to handle any cluster of interrupts that go off at nearly the same time.

It is necessary to provide interrupts the ability to pass parameters to the background sync functions. This is often data that must be sampled coincident with the interrupt in order to capture the correct external state. To avoid problems with variable life times, data is passed by copying it into a parameter area. Clearly, this is not a good strategy for passing a large amount of data. In those cases, it is necessary to manage memory between the background and the interrupts. No facilities are provided by the mechanisms to do this as it usually must to be constructed *ad hoc* to suit the particular needs of the data transfer. Usually it is sufficient to manage a pool in the background and use the synchronization mechanism to allow interrupt service to return memory to the pool when it is no longer needed.

For our purposes, it is only necessary to define some data structure that can be used by the interrupt service code to place data that will be delivered to the background function. The same considerations that were discussed for event parameters apply for sync parameters. So we use the same strategy.

96c ⟨*data types* 10⟩+≡ (149 152 155) ◁89c 97a▷

```
typedef EventParamType SyncParamType ;
```

Defines:
`SyncParamType`, used in chunk 97.
Uses `EventParamType` 23a.

When a sync function is invoked, it is actually passed a reference to its parameters.

97a ⟨*data types* 10⟩+≡ (149 152 155) ◁96c 97b▷

```
typedef SyncParamType *SyncParamRef ;
```

Defines:
`SyncParamRef`, used in chunks 60, 95–98, 109c, 110a, 129, and 145.
Uses `SyncParamType` 96c.

And so we can define the prototype for a sync function.

97b ⟨*data types* 10⟩+≡ (149 152 155) ◁97a

```
typedef void (*SyncFunc)(SyncParamRef) ;
```

Defines:
`SyncFunc`, used in chunks 95–98, 109c, 110a, 129, and 145.
Uses `SyncParamRef` 97a.

Each element of the sync queue is a pointer to the sync function and a place where the interrupt service code may place the values of the parameters.

97c ⟨*fgsync data types* 97c⟩≡ (150 153 156) 97d▷

```
typedef struct fgsyncblock {
    SyncFunc function ;
    SyncParamType params ;
} *FgSyncBlock ;
```

Defines:
`FgSyncBlock`, used in chunks 97–99, 102a, and 130a.
Uses `SyncFunc` 97b and `SyncParamType` 96c.

The sync queue is stored in an array and we use a couple of pointers to keep track of the head and tail.

97d ⟨*fgsync data types* 97c⟩+≡ (150 153 156) ◁97c

```
struct syncqueue {
    FgSyncBlock head ;
    FgSyncBlock tail ;
} ;
```

Uses `FgSyncBlock` 97c.

Following our usual pattern, we allocate storage for the sync queue entries as an array and for the control structure that tracks the queue boundaries within the sync queue entry array.

97e ⟨*sync queue* 97e⟩≡ (150 153 156)

```
static struct fgsyncblock
mechSyncQueueStorage[MECH_SYNCQUEUESIZE] ;
static struct syncqueue mechSyncQueue = {
    .head = mechSyncQueueStorage,
    .tail = mechSyncQueueStorage,
} ;
```

Defines:
`mechSyncQueue`, used in chunks 98 and 99.

There are only three operations on the sync queue. First we must be able to determine if the queue is empty or not. Emptiness is determined when the queue head is equal to the queue tail. For this type of queue operation, the head is where the next entry is removed and the tail is where the next entry is inserted.

98a ⟨*sync queue func* 98a⟩≡ (150 153 156) 98b ▷

```
static inline
bool
syncQueueEmpty(void)
{
    return mechSyncQueue.head == mechSyncQueue.tail ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `mechSyncQueue` 97e.

Inserting a function into the synchronization queue is an activity that may only be performed from interrupt service level. *N.B.* that this function does *not* implement a critical section around access to the sync queue.

98b ⟨*sync queue func* 98a⟩+≡ (150 153 156) ◁ 98a 99 ▷

```
static inline
SyncParamRef
syncQueuePut(
    SyncFunc f,
    bool fatal)
{
    FgSyncBlock tail = mechSyncQueue.tail ;
    if (++mechSyncQueue.tail >=
            mechSyncQueueStorage + MECH_SYNCQUEUESIZE) {
        mechSyncQueue.tail = mechSyncQueueStorage ;
    }
    if (syncQueueEmpty()) {
        if (fatal) {
            mechFatalError(mechSyncOverflow) ;
        }
        return NULL ;
    }

    tail->function = f ;
    return &tail->params ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `FgSyncBlock` 97c, `mechSyncQueue` 97e, `SyncFunc` 97b, and `SyncParamRef` 97a.

Insertion happens at the tail. Incrementing the tail pointer must account for wrapping around the array boundary. Overflow is detected as the queue being empty after the insertion is made. This logic has the effect of consuming one of the queue storage slots in order to detect overflow and this will need to be accounted for in sizing the sync queue storage. The `fata` argument determines if queue overflow is a fatal system error or not. This allow for dealing with sync requests that can be ignore if it would cause a sync queue overflow. Note that the sync queue slot is not modified until after overflow is detected. This insures that should an error happen during development, the queue can be examined and will be in a consistent state.

The consumer of the sync queue entries is the main loop.

99 ⟨*sync queue func* 98a⟩+≡ (150 153 156) ◁98b

```
static inline
FgSyncBlock
syncQueueGet(void)
{
    FgSyncBlock head ;

    beginCriticalSection() ;
    if (syncQueueEmpty()) {
        head = NULL ;
    } else {
        head = mechSyncQueue.head ;
        if (++mechSyncQueue.head >=
                mechSyncQueueStorage + MECH_SYNCQUEUESIZE) {
            mechSyncQueue.head = mechSyncQueueStorage ;
        }
    }
    endCriticalSection() ;

    return head ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `FgSyncBlock` 97c and `mechSyncQueue` 97e.

Since the main loop runs with interrupts enabled, obtaining a sync queue entry must be done in a critical section. We remove entries from the head of the queue. Otherwise, the only complexity in the code is to account for the array storage wrap around.

7 Initialization

Finally, we must consider initialization. The main loop delegates the initialization to a single function, `mechInit`. We divide the initialization into two parts, that needed for the mechanisms and that for the application.

Applications must supply a function to initialize hardware and one to initialize the domains in the application.

100a ⟨*external scoped functions* 15⟩+≡ (149 152 155) ◁96a

```
extern void sysDeviceInit(void) ;
extern void sysDomainInit(void) ;
```

Initialization is platform specific and divided into parts that initialize the critical section mechanisms, the timer used for delayed events and then a catch all for other platform specific interfaces. The sequence of initialization insures that by the time application specific initialization is called, the mechanisms are ready and capable of generating events.

100b ⟨*external test functions* 6⟩+≡ (149 152 155) ◁6 102b▷

```
extern void mechInit(void) ;
```

100c ⟨*initialization* 100c⟩≡ (150 153 156)

```
MECH_TEST_STATIC
MECH_TEST_INLINE
void
mechInit(void)
{
    sysPlatformInit() ;
    mechEventInit() ;
    initCriticalSection() ;
    sysTimerInit() ;
    sysDeviceInit() ;
    sysDomainInit() ;
}
```

8 Testing

The internals of the mechanisms are meant to be closed to application code. However, we must be able to support testing of the code and that requires some access to the internals. Here we discuss that aspects of the code base that are used for testing.

101a ⟨*conditional compilation* 80⟩+≡ (149 152 155) ◁89b

```
   * If the symbol MECH_TEST is defined to the preprocessor,
   * then code supporting testing the mechanisms will be
   * included in the object file.
```

We mainly provide the ability to access portions of the main loop processing for testing. The test program can then generate events, dispatch them and examine the consequences. To cope with delayed events and other external synchronizations we also need access to the sync queue and we need to be able to synchronize things. These considerations also mean that some functions will have to be made external in scope.

101b ⟨*conditional defines* 101b⟩≡ (150 153 156)

```
  #ifdef MECH_TEST
  #   define MECH_TEST_INLINE
  #   define MECH_TEST_STATIC
  #else
  #   define MECH_TEST_INLINE  inline
  #   define MECH_TEST_STATIC  static
  #endif /* MECH_TEST */
```

Uses `inline` 17 18a 26a 26b 27a 27b 27c 30a 31a 34b 37a 37b 37c 38 44 45a 84 85 86 98a 98b 99 103b 104a 104b 105 105 122 123a 123a 125a 125a 136 136 136 138a 138a 139.

First we start with executing sync functions.

101c ⟨*main loop components declaration* 101c⟩≡ (149 152 155)

```
  extern bool mechInvokeOneSyncFunc(void) ;
```

This function removes one function from the sync queue and executes it. The return value indicates whether or not a sync function was executed (*i.e.* if the sync queue was not empty).

102a ⟨*main loop sync components* 102a⟩≡ (150 156)
```
MECH_TEST_STATIC
MECH_TEST_INLINE
bool
mechInvokeOneSyncFunc(void)
{
    bool didOne ;
    FgSyncBlock blk = syncQueueGet() ;
    if (blk && blk->function) {
        blk->function(&blk->params) ;
        didOne = true ;
    } else {
        didOne = false ;
    }
    return didOne ;
}
```
Uses `FgSyncBlock` 97c.

Note that this function is made static in scope and inlined when we are not compiling for testing.

It is also convenient for testing purposes to be able to dispatch a single event and then regain control of the execution flow. So we define a function to accomplish that.

102b ⟨*external test functions* 6⟩+≡ (149 152 155) ◁100b 118b▷
```
extern bool mechDispatchOneEvent(void) ;
```

This function may be used to dispatch a single event and returns a boolean value indicated whether or not an event was actually dispatched.

102c ⟨*main loop components* 102c⟩≡ (150 156)
```
MECH_TEST_STATIC
MECH_TEST_INLINE
bool
mechDispatchOneEvent(void)
{
    bool didOne = !eventQueueEmpty(&eventQueue) ;
    if (didOne) {
        MechEcb ecb = eventQueue.next ;
        eventQueueRemove(ecb) ;
        mechDispatch(ecb) ;
    }
    return didOne ;
}
```
Uses `eventQueue` 28c and `MechEcb` 24.

9 POSIX Specific Interfaces

At this point we have reached the end of the generic code and must now begin to account for the platform differences in the way timing and asynchronous execution is handled. This software architecture can be run on a conventional UNIX platform. This includes Linux, Mac OS X and even Cygwin. The primary purpose of making the mechanisms run in a POSIX environment is simulation. Often, a domain can be executed for testing and simulation purposes on a conventional computer more easily than in the target environment. With I/O and disk storage, testing and tracing logic is often much easier.

9.1 POSIX Critical Sections

We start with a discussion of how to implement a critical section in POSIX. In POSIX, the *signal* is the mechanism of asynchronous execution. There are times when we must insure that execution is *not* interrupted by asynchronous signal execution. The functions in this section accomplish that.

The technique here is to maintain a signal mask of all the signals that are under control of the mechanisms. This signal mask can then be used to control signal execution as necessary. Note that an application can call low level signal handling primitives and manage subsets of signals outside of the mechanisms. This is definitely discouraged.

103a ⟨*posix critical section* 103a⟩≡ (150) 103b ▷

```
static sigset_t mechSigMask ;
```

Defines:
`sigset_t`, used in chunks 106 and 119.

It is necessary to initialize our managed signal mask to be empty.

103b ⟨*posix critical section* 103a⟩+≡ (150) ◁ 103a 104a ▷

```
static inline
void
initCriticalSection(void)
{
    sigemptyset(&mechSigMask) ;
}
```

Defines:
`inline`, used in chunk 101b.

Starting a critical section just means that we must block all the managed signals.

104a ⟨*posix critical section* 103a⟩+≡ (150) ◁103b 104b▷

```
static inline
void
beginCriticalSection(void)
{
    if (sigprocmask(SIG_BLOCK, &mechSigMask, NULL) != 0) {
        mechFatalError(mechSignalOpFailed, strerror(errno)) ;
    }
}
```

Defines:
`inline`, used in chunk 101b.

The end of a critical section is equally easily accomplished by unblocking the managed signals.

104b ⟨*posix critical section* 103a⟩+≡ (150) ◁104a

```
static inline
void
endCriticalSection(void)
{
    if (sigprocmask(SIG_UNBLOCK, &mechSigMask, NULL) != 0) {
        mechFatalError(mechSignalOpFailed, strerror(errno)) ;
    }
}
```

Defines:
`inline`, used in chunk 101b.

9.2 POSIX Timing Interfaces

In this section we present the timing interface for POSIX systems. In this timing scheme, the interval timer that measures real time is used and notifications of elapsed time arrive via SIGALRM.

In an embedded system, time is usually measured in units of clock ticks, where the amount of real time represented by a clock tick will vary from system to system. It is useful then to run the delayed event queue in hardware device units rather than a conventional time measure so that we may avoid the conversion computation each time the delayed event queue timing is started or stopped.

So, we introduce a couple of functions to convert between clock ticks and milliseconds and *vice versa*. Since the POSIX interface operates at a higher level, in the POSIX case there is no transformation. Note that the is a bit of data type slight of hand going on here. We use the `MechDelayTime` type to hold values of milliseconds and clock ticks. In practice this is not a problem, but the type will have to be chosen to account for the largest values held either of the uses of the data type.

105 ⟨*posix delayed event helper* 105⟩≡ (150)

```
static inline
MechDelayTime
mechMsecToTicks(
    MechDelayTime msec)
{
    return msec ;
}
static inline
MechDelayTime
mechTicksToMsec(
    MechDelayTime ticks)
{
    return ticks ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MechDelayTime` 40.

106 ⟨*posix timer services* 106⟩≡ (150) 107 ▷

```
static void
sysTimerMask(void)
{
    /*
     * Make sure SIGALRM does not go off.
     */
    sigset_t mask ;
    sigemptyset(&mask) ;
    sigaddset(&mask, SIGALRM) ;
    if (sigprocmask(SIG_BLOCK, &mask, NULL) != 0) {
        mechFatalError(mechSignalOpFailed, strerror(errno)) ;
    }
}

static void
sysTimerUnmask(void)
{
    /*
     * Allow SIGALRM to notify us.
     */
    sigset_t mask ;
    sigemptyset(&mask) ;
    sigaddset(&mask, SIGALRM) ;
    if (sigprocmask(SIG_UNBLOCK, &mask, NULL) != 0) {
        mechFatalError(mechSignalOpFailed, strerror(errno)) ;
    }
}
```

Uses sigset_t 103a.

To start a timer we supply the number of ticks we want to expire before we are notified.

107 ⟨*posix timer services* 106⟩+≡ (150) ◁106 108▷

```
static void
sysTimerStart(
    MechDelayTime time)
{
    struct itimerval delayedEventTimer ;

    delayedEventTimer.it_interval.tv_sec = 0 ;
    delayedEventTimer.it_interval.tv_usec = 0 ;
    delayedEventTimer.it_value.tv_sec = time / 1000 ;
    delayedEventTimer.it_value.tv_usec = (time % 1000) * 1000 ;

    if (setitimer(ITIMER_REAL, &delayedEventTimer, NULL) != 0) {
        mechFatalError(mechTimerOpFailed, strerror(errno)) ;
    }
    sysTimerUnmask() ;
}
```

Uses `MechDelayTime` 40.

The code initializes the *real* interval timer to use in servicing the delayed event queue. We set the `it_interval` member, which represents the next value to be loaded into the timer, to 0. Then when the time given by the `it_value` member expires the timer is stopped. The system timer is specified in microseconds and our time value is in milliseconds, so some conversion must be performed. Upon expiration, SIGALRM is generated. Notice that we exit the function with SIGALRM unblocked.

Stopping the timer returns the amount of time that had not elapsed.

108 ⟨*posix timer services* 106⟩+≡ (150) ◁107 109a▷

```
static MechDelayTime
sysTimerStop(void)
{
    sysTimerMask() ;
    /*
     * Fetch the remaining time.
     */
    struct itimerval delayedEventTimer ;
    if (getitimer(ITIMER_REAL, &delayedEventTimer) != 0) {
        mechFatalError(mechTimerOpFailed, strerror(errno)) ;
    }
    /*
     * Convert the returned time into milliseconds.
     */
    MechDelayTime remain =
            delayedEventTimer.it_value.tv_sec * 1000 +
            delayedEventTimer.it_value.tv_usec / 1000 ;
    /*
     * Set the current timer value to zero to turn it off.
     */
    memset(&delayedEventTimer, 0, sizeof(delayedEventTimer)) ;
    if (setitimer(ITIMER_REAL, &delayedEventTimer, NULL) != 0) {
        mechFatalError(mechTimerOpFailed, strerror(errno)) ;
    }

    return remain ;
}
```

Uses `MechDelayTime` 40.

Stopping the timer must make sure that SIGALRM does not expire during the process of getting things stopped.

Since the timing services use SIGALRM, the signal registration function is used to insure that SIGALRM is serviced.

109a ⟨*posix timer services* 106⟩+≡ (150) ◁108 109b▷
```
static void
sysTimerExpire(
    int signum)
{
    MechDelayTime nextTime = mechTimerExpireService() ;
    if (nextTime != 0) {
        sysTimerStart(nextTime) ;
    }
}
```
Uses `MechDelayTime` 40.

109b ⟨*posix timer services* 106⟩+≡ (150) ◁109a
```
static void
sysTimerInit(void)
{
    mechRegisterSignal(SIGALRM, sysTimerExpire) ;
}
```

9.3 POSIX Async Execution Interface

The POSIX view of a process includes the notion of *signals*. Signals are a form of asynchronous execution, and reasonably correspond to the interrupts of a bare metal system. As we have seen in the timer services above, we can use signals to access a variety of services on a POSIX system.

In this section we fill out the asynchronous execution interfaces using signals. As we will see, POSIX systems also require that you deal with their I/O interface in order to properly handle execution sequencing. For now, we present an interface that allows an application to deal with the asynchronous aspects of signals. As expected, the interface allows an application to register a sync function that will be executed at the first safe opportunity after the signal has expired.

109c ⟨*posix async interface* 109c⟩≡ (150) 110a▷
```
SyncParamRef
mechSyncRequest(
    SyncFunc f)
{
    return syncQueuePut(f, true) ;
}
```
Uses `SyncFunc` 97b and `SyncParamRef` 97a.

The alternate interface is also easily implemented.

110a ⟨*posix async interface* 109c⟩+≡ (150) ◁109c 111▷

```
SyncParamRef
mechTrySyncRequest(
    SyncFunc f)
{
    return syncQueuePut(f, false) ;
}
```

Uses `SyncFunc` 97b and `SyncParamRef` 97a.

Applications can register a signal function that will be called ultimately be called after the signal triggers.

110b ⟨*posix external functions* 110b⟩≡ (149) 112b▷

```
typedef void (*SignalFunc)(int) ;
extern void
mechRegisterSignal(
    int sigNum,
    SignalFunc func) ;
```

Defines:
`SignalFunc`, used in chunk 111.

The arguments are simply, `sigNum`, the number of the signal being registered and `func`, a pointer to a sync function that will be called. If `func` is `NULL`, then the signal's behavior is reset to its default behavior.

111 ⟨*posix async interface* 109c⟩+≡ (150) ◁110a

```
void
mechRegisterSignal(
    int sigNum,
    SignalFunc func)
{
    assert(sigNum > 0) ;

    struct sigaction action ;
    if (func) {
        action.sa_handler = func ;
        sigaddset(&mechSigMask, sigNum) ;
    } else {
        action.sa_handler = SIG_DFL ;
        sigdelset(&mechSigMask, sigNum) ;
    }
    sigfillset(&action.sa_mask) ;
    action.sa_flags = 0 ;

    int sigresult = sigaction(sigNum, &action, NULL) ;
    if (sigresult != 0) {
        mechFatalError(mechSignalOpFailed, strerror(errno)) ;
    }
}
```

Uses `SignalFunc` 110b.

We set up signal handlers to run uninterrupted by other signals. This is accomplished by filling the `sa_mask` member of the `sigaction` structure. This is simplifies keeping track of what is going on.

9.4 POSIX I/O Interface

On POSIX platforms, the mechanisms must also supply services to handle I/O. The reason for this is that there are two means of awakening a sleeping process, receiving a signal and servicing a I/O file descriptor.[3] On bare metal systems, I/O is frequently accomplished on an *ad hoc* basis and the single mechanism of the sync queue is usually sufficient. However, POSIX makes a distinction between signals and I/O operations on file descriptors and an interface needs to be provided to deal with servicing file descriptors that require attention.

We model this interface after the one for signals. The idea is that a set of callback functions can be registered on a file descriptor for reading, writing or an exception. When the condition is satisfied the callback is invoked. So we must define an I/O callback function as:

112a ⟨*posix data types* 112a⟩≡ (149)
```
typedef void (*FDServiceFunc)(int) ;
```
Defines:
`FDServiceFunc`, used in chunks 112b, 114a, and 116.

When the callback is invoked, it is passed the value of the file descriptor that requires service.

The mechanisms provide two functions for I/O. One registers the callbacks for a file descriptor and the other removes a file descriptor from consideration.

112b ⟨*posix external functions* 110b⟩+≡ (149) ◁110b 113▷
```
extern void
mechRegisterFDService(
    int fd,
    FDServiceFunc readService,
    FDServiceFunc writeService,
    FDServiceFunc exceptService) ;
```
Uses `FDServiceFunc` 112a.

[3] There are other ways to integrate signals and I/O in POSIX systems. The use of a `pselect` based approach is a design decision.

The function, `mechRegisterFDService`, registers callbacks for a file descriptor.

`fd` A file descriptor as returned from `open`, `socket` or any other system calls that create file descriptors.

`readService` A pointer to a callback function that will be registered for the file descriptor and invoked when the file descriptor is readable or `NULL` if no callback is registered.

`writeService` A pointer to a callback function that will be registered for the file descriptor and invoked when the file descriptor is writable or `NULL` if no callback is registered.

`exceptService` A pointer to a callback function that will be registered for the file descriptor and invoked when the file descriptor is in an exception condition or `NULL` if no callback is registered. In practice, exception conditions are used only for reading OOB (out of bands) data on a TCP socket.

The corresponding remove function has the following interface.

113 ⟨*posix external functions* 110b⟩+≡ (149) ◁112b

```
extern void
mechRemoveFDService(
    int fd,
    bool rmRead,
    bool rmWrite,
    bool rmExcept) ;
```

`fd` A file descriptor as returned from `open`, `socket` or any other system calls that create file descriptors.

`rmRead` A boolean indicated whether or not the file descriptor should have its read callback unregistered.

`rmWrite` A boolean indicated whether or not the file descriptor should have its write callback unregistered.

`rmExcept` A boolean indicated whether or not the file descriptor should have its exception callback unregistered.

The implementation of these two functions requires some internal data structures to track the file descriptor sets. File descriptor sets are handed to `pselect` to indicate how a process is to be awakened for I/O servicing.

To track the callback functions we need a data structure.

114a ⟨*posix io data* 114a⟩≡ (150) 114b ▷

```
typedef struct fdservicemap {
    bool set ;
    FDServiceFunc read ;
    FDServiceFunc write ;
    FDServiceFunc except ;
} *FDServiceMap ;
```

Defines:
`FDServiceMap`, used in chunks 116, 117, and 119.
Uses `FDServiceFunc` 112a.

`set` A boolean indicating whether or not the entry is in use.

`read` A pointer to the read callback registered for the file descriptor or `NULL` if no callback is registered.

`write` A pointer to the write callback registered for the file descriptor or `NULL` if no callback is registered.

`except` A pointer to the exception callback registered for the file descriptor or `NULL` if no callback is registered.

Following our familiar pattern, we define an array of mapping entries that defines a pool for storing the entries that map file descriptor state to callbacks. This array is indexed by file descriptor value.

114b ⟨*posix io data* 114a⟩+≡ (150) ◁ 114a 115a ▷

```
static struct fdservicemap mechFDServicePool[FD_SETSIZE] ;
```

Defines:
`mechFDServicePool`, used in chunks 116, 117, and 119.

The value of `FD_SETSIZE` is determined by the system and is the maximum number of file descriptors that can be in a `fd_set` given to `select`.

One complication of using `pselect` as a means of registering intent on multiple file descriptors is that you must keep track of the largest value of a file descriptor in the set handed to `pselect`. This is an argument to `pselect` (and `select`). Fortunately, UNIX file descriptors operate in a rather predictable manner. Each process has file descriptors 0, 1, and 2 open when the process is started. Creating a new file descriptor (*e.g.* by opening a file) will allocate the next largest unused file descriptor number. This rule applies to the three file descriptors opened by default for a process. So, for example, closing file descriptor 2 and then opening a new file will result in file descriptor 2 being reused. All this makes tracking the maximum file descriptor number relatively easy. We only need a single integer variable.

115a ⟨*posix io data* 114a⟩+≡ (150) ◁114b 115b▷

```
static int mechMaxFD = -1 ;
```

Defines:
`mechMaxFD`, used in chunks 116, 117, and 119.

Since `mechMaxFD` holds the maximum value of the file descriptors that have been registered, the value -1 indicates that there are no registered file descriptors.

We need variables to hold the three sets of file descriptors needed by `pselect`.

115b ⟨*posix io data* 114a⟩+≡ (150) ◁115a

```
static fd_set mechReadFDS ;
static fd_set mechWriteFDS ;
static fd_set mechExceptFDS ;
```

Defines:
`fd_set`, used in chunk 119.

Finally, we can talk about the implementation of the I/O registration functions.

116 ⟨*posix io interface* 116⟩≡ (150) 117 ▷

```
void
mechRegisterFDService(
    int fd,
    FDServiceFunc readService,
    FDServiceFunc writeService,
    FDServiceFunc exceptService)
{
    assert(fd >= 0 && fd < FD_SETSIZE) ;
    FDServiceMap fds = mechFDServicePool + fd ;

    fds->read = readService ;
    if (readService) {
        FD_SET(fd, &mechReadFDS) ;
        fds->set = true ;
    } else {
        FD_CLR(fd, &mechReadFDS) ;
    }

    fds->write = writeService ;
    if (writeService) {
        FD_SET(fd, &mechWriteFDS) ;
        fds->set = true ;
    } else {
        FD_CLR(fd, &mechWriteFDS) ;
    }

    fds->except = exceptService ;
    if (exceptService) {
        FD_SET(fd, &mechExceptFDS) ;
        fds->set = true ;
    } else {
        FD_CLR(fd, &mechExceptFDS) ;
    }

    if (fds->read == NULL && fds->write == NULL && fds-> except == NULL) {
        if (fds->set && fd >= mechMaxFD) {
            --mechMaxFD ;
        }
        fds->set = false ;
    } else if (fds->set && fd > mechMaxFD) {
        mechMaxFD = fd ;
    }
```

```
  }
```

Uses `FDServiceFunc` 112a, `FDServiceMap` 114a, `mechFDServicePool` 114b, and `mechMaxFD` 115a.

The function simply tests the various callback functions and if they are not `NULL`, then the file descriptor is added to the appropriate set. As written, `mechRegisterFDService` may be used to modify file descriptors already registered. The last bit of logic is there in case `mechRegisterFDService` is used to effectively remove the file descriptor by supplying three `NULL` callback function pointers. We must also account for the maximum file descriptor value that has been registered.

Removing a file descriptor from consideration is also straight forward.

117 ⟨*posix io interface* 116⟩+≡ (150) ◁116 118a▷

```
  void
  mechRemoveFDService(
      int fd,
      bool rmRead,
      bool rmWrite,
      bool rmExcept)
  {
      assert(fd >= 0 && fd < FD_SETSIZE) ;
      FDServiceMap fds = mechFDServicePool + fd ;

      if (rmRead) {
          fds->read = NULL ;
          FD_CLR(fd, &mechReadFDS) ;
      }

      if (rmWrite) {
          fds->write = NULL ;
          FD_CLR(fd, &mechWriteFDS) ;
      }

      if (rmExcept) {
          fds->except = NULL ;
          FD_CLR(fd, &mechExceptFDS) ;
      }

      if (fds->read == NULL && fds->write == NULL && fds-> except == NULL &&
              fd >= mechMaxFD) {
          mechMaxFD = fd - 1 ;
      }
  }
```

Uses `FDServiceMap` 114a, `mechFDServicePool` 114b, and `mechMaxFD` 115a.

We also need something to initialize the file descriptor sets that we are maintaining.

118a ⟨*posix io interface* 116⟩+≡ (150) ◁117

```
static void
fdServiceInit(void)
{
    FD_ZERO(&mechReadFDS) ;
    FD_ZERO(&mechWriteFDS) ;
    FD_ZERO(&mechExceptFDS) ;
}
```

9.5 POSIX Suspending Execution

The main loop detects when there is nothing left to do and suspends execution. Here we present how that suspension happens for the POSIX version of the mechanisms.

This design is based on using `pselect` to suspend a process until either a signal occurs or a file descriptor requires service. The `mechWait` function is called by the main loop when there is no work currently to be done. It is invoked inside of a critical section. This is an important entry condition for `mechWait`. In the POSIX case, this means that `mechWait` must be invoked with all the registered signals blocked. We then use `pselect` to atomically enable all signals and block the process.

118b ⟨*external test functions* 6⟩+≡ (149 152 155) ◁102b

```
extern void mechWait(void) ;
```

119 ⟨*posix suspend execution* 119⟩≡ (150)

```
MECH_TEST_STATIC
void
mechWait(void)
{
    beginCriticalSection() ;
    if (syncQueueEmpty()) {
        /*
         * Copy the file descriptor sets since "pselect"
         * modifies them in place upon return.
         */
        fd_set readfds ;
        memcpy(&readfds, &mechReadFDS, sizeof(readfds)) ;
        fd_set writefds ;
        memcpy(&writefds, &mechWriteFDS, sizeof(writefds)) ;
        fd_set exceptfds ;
        memcpy(&exceptfds, &mechExceptFDS, sizeof(exceptfds)) ;
        /*
         * Allow all the signals during the select. We
         * assume that when we enter this function, the
         * registered signals are masked.
         */
        sigset_t mask ;
        sigemptyset(&mask) ;
        /*
         * "mechMaxFD" holds the maximum value of any
         * registered file descriptor. We must add one to
         * get the number of file descriptors "pselect" is
         * to consider.
         */
        int r = pselect(mechMaxFD + 1, &readfds, &writefds,
                &exceptfds, NULL, &mask) ;
        if (r == -1) {
            if (errno != EINTR) {
                mechFatalError(mechSelectWaitFailed,
                        strerror(errno)) ;
            }
            /*
             * Got a signal while waiting. We go back to the
             * main loop on the assumption that something
             * has been placed in the sync queue.
             */
        } else {
            /*
             * Dispatch the service functions for the file
             * descriptors.
```

```
                 */
                FDServiceMap s = mechFDServicePool ;
                for (int fd = 0 ; r > 0 && fd <= mechMaxFD ;
                        ++fd, ++s) {
                    /*
                     * Do exceptions first. This is only
                     * important for sockets, but without going
                     * first the OOB data processing won't
                     * work.
                     */
                    if (FD_ISSET(fd, &exceptfds)) {
                        assert(s->except != NULL) ;
                        s->except(fd) ;
                        --r ;
                    }
                    if (FD_ISSET(fd, &readfds)) {
                        assert(s->read != NULL) ;
                        s->read(fd) ;
                        --r ;
                    }
                    if (FD_ISSET(fd, &writefds)) {
                        assert(s->write != NULL) ;
                        s->write(fd) ;
                        --r ;
                    }
                }
            }
        }
        endCriticalSection() ;
}
```

Uses `fd_set` 115b 115b 115b, `FDServiceMap` 114a, `mechFDServicePool` 114b, `mechMaxFD` 115a, and `sigset_t` 103a.

By far, most of the work in `mechWait` is to deal with the file descriptor status changes. The file descriptor sets must be copied before being handed to `pselect` since it modifies them in place. After we determine that it was a file descriptor status change that caused us to wake up, we must go through and find all file descriptors that had a status change and invoke the callback function.

The way that we are using `pselect` in this circumstance may seem a bit backwards. Upon entry to `mechWait`, we start a critical section where the registered signals are blocked. The signal mask given to `pselect` is empty, meaning that `pselect` will allow all signals while the process sleeps. Upon the return from `pselect` we will be back to the state where the registered signals are blocked. Thus we avoid the race condition where we have determined that the sync queue is empty, but asynchronous execution that might affect the sync queue arrives before we can put the process to sleep. This is exactly the type of race condition `pselect` is used to prevent.

It is also worth noting we do *not* use any time out in the `pselect` invocation. All timing is done via delayed events, and they are signalled via `SIGALRM` and managed on the delayed event queue as discussed before (p. 39).

9.6 POSIX Initialization

Here we present the POSIX version of the required internal initialization.

121 ⟨*posix initialization* 121⟩≡ (150)
```
static void
sysPlatformInit(void)
{
    fdServiceInit() ;
}
```

9.7 POSIX Compilation

The POSIX version of this code is tested under Linux. There are some significant dependencies in the code on specific system functionality, especially for synchronization and suspending execution. Recent versions of `glibc` use a set of *feature test macros* to control how definitions are exposed in the system header files. Further, some of the mechanisms code depend upon *C99* capablity. So, to insure proper compilation under `gcc` the following compiler options should be used:

- `--std=c99`
- `-D_POSIX_C_SOURCE=200112L`

You may need to adjust these options for your particular version of `gcc`.

10 ARM Cortex-M3 Specific Interfaces

The ARM® Cortex-M3® core is a relatively recent entry into the low end micro-controller world. It is significant that as an ARM core it brings a much more modern architecture than typically found in this realm. Most important for our concerns here are a set of standard peripherals for handling interrupts, debugging and other system related issues. This significantly eases the problem of porting low level system code to the different chips that have Cortex-M3 cores. The reader is referred to the excellent book [6] for complete description of the Cortex-M3 core.

However, a core is just a core and all real chips that incorporate a Cortex-M3 core have many other peripherals. Inevitably, this causes chip specific differences that have to be accounted for. This variability is somewhat compensated for by ARM having come out with the Cortex Micro-controller Software Interface Standard (CMSIS) effort to standardize the naming and “C” language header files for accessing the both the ARM defined core and vendor specific peripherals. Vendors now supply header files that deal with the peripherals that they include on the chips as well as the standard parts of the Cortex-M3 core. We use the CMSIS interfaces in this program.

10.1 Cortex-M3 Critical Sections

The three functions needed to implement critical sections are straight forward in the Cortex-M3 implementation. We use the initialization function to set the priority of the PendSV exception to be the lowest priority (highest priority number) of any exception. It is critical in designing your priority scheme insure that PendSV is the lowest priority exception.

122 ⟨*cortex-m3 critical section* 122⟩≡ (153) 123a ▷

```
static inline
void
initCriticalSection(void)
```

```
{
    #define MECH_PENDSV_PRIORITY     0xff
    NVIC_SetPriority(PendSV_IRQn, MECH_PENDSV_PRIORITY) ;
}
```

Defines:
`inline`, used in chunk 101b.

As is usual in bare metal environments, critical sections are implemented by disabling interrupts. Note that in the Cortex-M3, disabling interrupts still allows the bus fault and other high priority exception to occur.

123a ⟨*cortex-m3 critical section* 122⟩+≡ (153) ◁122

```
static inline
void
beginCriticalSection(void)
{
    __disable_irq() ;
}
static inline
void
endCriticalSection(void)
{
    __enable_irq() ;
}
```

Defines:
`inline`, used in chunk 101b.

10.2 Cortex-M3 Timing Interfaces

The timing interfaces for the Cortex-M3 are more complicated than for POSIX simply because there is less standard interface to depend upon. It is tempting to try to use the SysTick counter for delayed event timing. However, this may not be possible depending upon the details of the way SysTick is clocked.

1. Often SysTick is clocked by the same clock as the processor core. This is often very fast, in the megahertz range.
2. The SysTick counter is only 24 bits. That combined with a fast clock yields a very limited range of timing.

It is clear that SysTick was designed for periodic timing interrupts of the type we are explicitly trying to avoid. However, we give a SysTick implementation here as a default which should also serve as an example for how to use other timing resources for delayed events.

123b ⟨*cm3 conditional compilation* 123b⟩≡ (152) 124a ▷

```
 * When compiling for the Cortex-M3, defining the symbol
 * CM3_USE_SYSTICK indicates that the SysTick clock
 * will be used for delayed event timing.
```

In low power systems, 32 KiHz clocks are common since low power oscillators of this frequency are readily available. One convenient clocking scheme is to divide 32 KiHz down by 8 to yield a 4 KiHz frequency which is approximately 244 microseconds per tick. This clocking combined with a 24 bit timer register gives reasonable ranges at resolutions good enough for most application level timing. Remember, delayed events are not designed for high rate precise timing. If that is required, you will need to dedicate a timing resource to such needs.

124a ⟨*cm3 conditional compilation* 123b⟩+≡ (152) ◁123b

```
 * When compiling for the Cortex-M3, the value of the symbol
 * MECH_TIMER_FREQUENCY gives the number of clock ticks
 * in one second. The default value is 4096.
```

It is necessary to set up some constants to define the conversion from clock ticks to milliseconds, the units of delayed event timing.

124b ⟨*cortex-m3 delayed event helper* 124b⟩≡ (153) 125a▷

```
#ifdef CM3_USE_SYSTICK
extern uint32_t SystemCoreClock ;
#   define MECH_TIMER_FREQUENCY     SystemCoreClock
#else
#   ifndef MECH_TIMER_FREQUENCY
#       define MECH_TIMER_FREQUENCY     4096UL
#   endif /* MECH_TIMER_FREQUENCY */
#endif /* CM3_USE_SYSTICK */

#ifndef MECH_MAX_CLOCK_TICKS
#   ifdef CM3_USE_SYSTICK
#       define MECH_MAX_CLOCK_TICKS     0x00ffffffUL
#   else
#       define MECH_MAX_CLOCK_TICKS     (UINT32_MAX - 1)
#   endif /* CM3_USE_SYSTICK */
#endif /* MECH_MAX_CLOCK_TICKS */

#define MECH_MAX_MSEC_DELAY\
        ((MECH_MAX_CLOCK_TICKS / MECH_TIMER_FREQUENCY) * 1000UL)
```

Defines:
MECH_MAX_MSEC_DELAY, used in chunks 125a and 138a.
Uses MECH_MAX_CLOCK_TICKS 137b.

Unlike the POSIX delayed event timing, the conversation from ticks the milliseconds and *vice versa* is not the identity mapping.

125a ⟨*cortex-m3 delayed event helper* 124b⟩+≡ (153) ◁124b

```
static inline
MechDelayTime
mechMsecToTicks(
    MechDelayTime msec)
{
    if (msec > MECH_MAX_MSEC_DELAY) {
        msec = MECH_MAX_MSEC_DELAY ;
    }
    /*
     * We must avoid overflow if the requested number of
     * msecs is large.
     */
    return msec < MECH_MAX_CLOCK_TICKS / MECH_TIMER_FREQUENCY ?
            ((msec * MECH_TIMER_FREQUENCY) + 1000UL / 2UL) / 1000UL :
            ((msec + 1000UL / 2UL) / 1000UL) * MECH_TIMER_FREQUENCY ;
}

static inline
MechDelayTime
mechTicksToMsec(
    MechDelayTime ticks)
{
    return ticks < MECH_MAX_CLOCK_TICKS / 1000UL ?
        ((ticks * 1000UL) + MECH_TIMER_FREQUENCY / 2UL) / MECH_TIMER_FREQUENCY :
        ((ticks + MECH_TIMER_FREQUENCY / 2UL) / MECH_TIMER_FREQUENCY) * 1000UL ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MECH_MAX_CLOCK_TICKS` 137b, `MECH_MAX_MSEC_DELAY` 124b 137b, and `MechDelayTime` 40.

For the Cortex-M3, the following functions must be provided to operated the delayed event timing. We provide default ones based on SysTick, but, as discussed above, you may need to provide a different timing resource from SysTick and, consequently, provide an implementation for that timing resource.

125b ⟨*cortex-m3 timing declarations* 125b⟩≡ (152)

```
extern void sysTimerInit(void) ;
extern void sysTimerMask(void) ;
extern void sysTimerUnmask(void) ;
extern void sysTimerStart(MechDelayTime) ;
extern MechDelayTime sysTimerStop(void) ;
```

Uses `MechDelayTime` 40.

The SysTick implementation is defined to be weak so that you can override them with your own functions.

126 ⟨*cortex-m3 timer services* 126⟩≡ (153)

```
__WEAK
void
sysTimerInit(void)
{
    SysTick->CTRL = 0 ;
}

__WEAK
void
sysTimerMask(void)
{
    SysTick->CTRL &= ~SysTick_CTRL_TICKINT_Msk ;
}

__WEAK
void
sysTimerUnmask(void)
{
    SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk ;
}

__WEAK
void
sysTimerStart(
    MechDelayTime time)
{
    SysTick->LOAD = time ;
    SysTick->VAL = 0 ;
    SysTick->CTRL =
            SysTick_CTRL_CLKSOURCE_Msk |
            SysTick_CTRL_ENABLE_Msk |
            SysTick_CTRL_TICKINT_Msk ;
}

__WEAK
MechDelayTime
sysTimerStop(void)
{
    SysTick->CTRL = 0 ;
    return SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk ?
            0 : SysTick->LOAD - SysTick->VAL ;
}
```

```
void
SysTick_Handler(void)
{
    beginCriticalSection() ;
    SysTick->CTRL = 0 ;
    MechDelayTime nextTime = mechTimerExpireService() ;
    if (nextTime != 0) {
        sysTimerStart(nextTime) ;
    }
    endCriticalSection() ;
}
```

Uses `__WEAK` 155 and `MechDelayTime` 40.

10.3 Cortex-M3 Async Execution Interface

The ARM Cortex-M3 has a very flexible and rather sophisticated exception processing architecture. In this context exception processing involves exceptions detected by the processor as well as external interrupts. The Cortex-M3 provides individually assignable priorities, priority groups and a well defined set of system exceptions. The full capabilities are beyond the scope of this paper and here we will only discuss those features of the Cortex-M3 exception processing scheme that we use.

If you examine the asynchronous execution rules of the STSA carefully, you can boil things down to a very few requirements.

1. Interrupts make their presence known to the background thread by posting a synchronization request using `mechSyncRequest()` or `mechTrySyncRequest`.
2. Synchronization requests are executed in preference to dispatching events. While a synchronization request is executing, interrupts *must* be allowed.
3. While a state action is executing, synchronization requests *must not* be executed and interrupts *must* be allowed.

Clearly, there is a three priority scheme inherent in these rules. Note that only interrupts are allowed to be preemptive. Executing synchronization requests and state actions do not preempt each other. Each must wait for the other to complete. In those processors (or as we say above in the case of a POSIX process) that only support two priorities in hardware, we execute the synchronization requests in the main loop. However, if there is additional capability in the hardware, we can take advantage of it. The idea is to execute synchronization requests in a higher priority processing state.

The processing prioritization capability of the Cortex-M3 architecture can be used to accomplish a multi-priority processing scheme. The feature of the Cortex-M3 architecture that maps very well to these rules is the Pend Service or **PendSV** exception. PendSV is intended to provide a means for interrupts to request additional service. The priority of the PendSV execution may be set independently of the priority of ordinary background processing. When the processing level is appropriate, the PendSV exception will execute. The key to making this scheme work is to assign the lowest possible priority (highest possible priority number) to the PendSV exception and to insure that the priority of all other exceptions is greater (smaller priority number) than that of PendSV. This means that PendSV exception handler will execute only if there are no there are no other exceptions pending and that higher priority exceptions (larger priority number) may preempt the PendSV exception. In particular this insures that interrupts are allowed during the sync request processing. To prevent sync requests from executing during the state action the base priority of execution is set equal to the priority of the PendSV exception thereby insuring that it does not get executed. Since all other exception priorities are higher, they will be allowed to execute during a state action.

First, we examine the code to perform a sync request.

129a ⟨*cortex-m3 async interface* 129a⟩≡ (153) 129b ▷

```
SyncParamRef
mechSyncRequest(
    SyncFunc f)
{
    beginCriticalSection() ;
    SyncParamRef params = syncQueuePut(f, true) ;
    endCriticalSection() ;
    SCB->ICSR = SCB_ICSR_PENDSVSET_Msk ;
    return params ;
}
```

Uses `SyncFunc` 97b and `SyncParamRef` 97a.

Recall that `mechSyncRequest` is only intended to be called from interrupt service level. We insert a critical section around `syncQueuePut`. This is necessary in case the interrupt prioritization has been set such that preemption of a higher priority interrupt is allowed. The main difference here is that, in addition to queuing the sync request, the PendSV exception is made pending. Since `mechSyncRequest` is only executed at interrupt service priority, PendSV is indeed only made pending here. The variation that does not fail on queue overflow is similar.

129b ⟨*cortex-m3 async interface* 129a⟩+≡ (153) ◁ 129a 130a ▷

```
SyncParamRef
mechTrySyncRequest(
    SyncFunc f)
{
    beginCriticalSection() ;
    SyncParamRef params = syncQueuePut(f, false) ;
    endCriticalSection() ;
    if (params) {
        SCB->ICSR = SCB_ICSR_PENDSVSET_Msk ;
    }
    return params ;
}
```

Uses `SyncFunc` 97b and `SyncParamRef` 97a.

Of course it is necessary to have an exception handler. This takes the place of the code segment that would otherwise appear in the main loop.

130a ⟨*cortex-m3 async interface* 129a⟩+≡ (153) ◁129b

```
void
PendSV_Handler(void)
{
    /*
     * Empty the foreground / background
     * synchronization queue.
     */
    for (FgSyncBlock blk = syncQueueGet() ; blk ;
            blk = syncQueueGet()) {
        if (blk->function) {
            blk->function(&blk->params) ;
        }
    }
}
```

Uses `FgSyncBlock` 97c.

So we see that the exception handler just empties the sync queue in a tight loop. Because it runs at the lowest priority and assuming that the priority grouping has been set up to allow preemption of higher priority interrupts, any external interrupts may preempt the exception handler. Again, a critical section around the queue access in `syncQueueGet` is necessary because of the preemption possibilities.

The last step is to insure that state actions are not preempted by the PendSV handler. We can accomplish that by raising the base priority of execution to match that of the PendSV handler. So it is necessary to modify `mechDispatchOneEvent` slightly.

130b ⟨*cortex-m3 main loop components* 130b⟩≡ (153)

```
MECH_TEST_STATIC
MECH_TEST_INLINE
bool
mechDispatchOneEvent(void)
{
    __set_BASEPRI(MECH_PENDSV_PRIORITY) ;
    bool didOne = !eventQueueEmpty(&eventQueue) ;
    if (didOne) {
        MechEcb ecb = eventQueue.next ;
        eventQueueRemove(ecb) ;
        mechDispatch(ecb) ;
    }
    __set_BASEPRI(0) ;
    return didOne ;
}
```

Uses `eventQueue` 28c and `MechEcb` 24.

By raising the priority to match that of PendSV, we allow higher priority interrupts to occur, but when they invoke `mechSyncRequest` the PendSV execution is made pending and does *not* preempt the state action execution because of the elevated base priority.

10.4 Cortex-M3 Suspending Execution

For the POSIX version of STSA, suspending execution was complicated by the fact that there are really two things that can require the attention of a program, signals and I/O on a file descriptor. There are usually no concepts like file descriptors on bare metal so suspending on a Cortex-M3 is much simpler. Indeed there is a built in instruction to do the hard work.

The more difficult task in a low power system is making sure that suspending the processor puts the system in a state of lowered power consumption from which it will awaken when an interrupt occurs. This can be very system specific, so we delegate two functions to the ready the system for powering down and then up again.

131a ⟨*cortex-m3 power declarations* 131a⟩≡ (152)

```
extern void sysPowerDown(void) ;
extern void sysPowerUp(void) ;
```

The `sysPowerDown()` function is invoked just before going to sleep and the `sysPowerUp()` function is invoked just after waking up. Usually, these functions must handle enabling / disabling clocking of the microprocessor core and peripherals. Both are invoked in a critical section. We provide empty defaults, weakly referenced that can be overridden as necessary.

131b ⟨*cortex-m3 power management* 131b⟩≡ (153)

```
__WEAK
void
sysPowerDown(void)
{
}
__WEAK
void
sysPowerUp(void)
{
}
```

Uses `__WEAK` 155.

Now we can see what waiting entails on a Cortex-M3.

132 ⟨*cortex-m3 suspend execution* 132⟩≡ (153)

```
MECH_TEST_STATIC
MECH_TEST_INLINE
void
mechWait(void)
{
    beginCriticalSection() ;
    if (syncQueueEmpty() &&
            eventQueueEmpty(&eventQueue)) {
        sysPowerDown() ;
        __WFI() ;
        sysPowerUp() ;
    }
    endCriticalSection() ;
}
```

Uses `eventQueue` 28c.

The test for whether or not both the sync and event queues are empty must be done in a critical section. This insures that no interrupt gets in and posts a sync request while we are trying to determine if there is any additional work to be done.

There is actually another way suspend the Cortex-M3, Wait For Event (WFE). WFE is usually used in multi-processor systems and we don't consider it here.

10.5 Cortex-M3 Exception Scenarios

We now have sufficient detail to step through some of the cases of waking up and going to sleep for a Cortex-M3 and show how that interacts with interrupts and the PendSV handler to implement foreground / background synchronization. This should help clarify the design decisions in the Cortex-M3 case. We will consider three common cases:

1. We are asleep and an interrupt arrives.
2. We are executing a sync function and an interrupt arrives.
3. We are executing a state action and an interrupt arrives.

Interrupt While Sleeping

If we are asleep, then execution has stopped at the WFI instruction in `mechWait`. As soon as an interrupt is pending, execution resumes at the instruction after the WFI. Before going to sleep, we were in a critical section, so the interrupt handler does *not* execute. However, as soon as we exit the critical section, the interrupt will be made active and execution of the interrupt handler will begin. At some point we will assume the interrupt handler invokes `mechSyncRequest()`. This will cause the PendSV exception to become pending. Since PendSV runs at the lowest priority, it will not preempt the interrupt handler.

As soon as the interrupt handler exits and assuming that there are no other interrupts pending, the processor will tail chain to the PendSV handler as it will be the highest priority pending exception. The PendSV handler empties the sync queue. Since it runs at the lowest priority, it may be preempted by interrupts that occur. We will assume that a state machine event was posted by the sync function. Assuming that the sync queue is emptied and no exceptions are pending, the PendSV handler exits returning the execution to the end of `mechWait` where it was originally preempted.

Upon the return from `mechWait`, the main loop looks to dispatch any queued events. If it finds an event, the base priority of execution is raised to be the same as the PendSV priority. This will insure that the event dispatch is not preempted by the PendSV exception and therefore no sync functions will be executed while the state action is running. However, since interrupts are higher in priority, they may preempt the state action execution.

Interrupt During Sync Function Execution

In the above scenario, if an interrupt arrives while executing a sync function, the PendSV exception is preempted. The interrupt may make calls to `mechSyncRequest()`, queuing another sync request. Upon return from the interrupt handler, the PendSV handler will resume and continue executing until the sync queue is empty. Given that each invocation of `mechSyncRequest()` causes PendSV to become a pending exception, the processor will tail chain back to the PendSV handler. At some point in time the sync queue will be empty and no interrupts will have queue any other sync requests. Execution resumes at base priority 0.

Interrupt During State Action Execution

During the execution of a state action the base priority of execution is raised to be the same as the PendSV exception. This insures that interrupts may preempt the state action but that any sync requests by the interrupt handler will *not* execute. When the state action returns and the event dispatch is complete, the base priority is set back to zero. If the interrupt invoked `mechSyncRequest()`, then PendSV will be pending and as soon as the base priority is lowered back to zero, background execution is preempted and the PendSV handler is invoked.

10.6 Cortex-M3 Initialization

Initializing a Cortex-M3 platform can be rather complicated. Each chip vendor typically has a different clocking scheme that must be set up and for the most part you are own your own here. The supplied function below is far to minimal to be of any real use in a a real system, but contains a couple of things that usually need to be set up on a Cortex-M3.

134 ⟨*cortex-m3 external scoped functions* 134⟩≡ (152)
```
extern void sysPlatformInit(void) ;
```

135 ⟨*cortex-m3 initialization* 135⟩≡ (153)

```
__WEAK
void
sysPlatformInit(void)
{
    /*
     * Set stack align, so ISR's are truly ordinary "C"
     * functions.
     */
    SCB->CCR |= SCB_CCR_STKALIGN_Msk ;
    /*
     * Deep sleep when we wait
     */
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk ;
}
```

Uses `__WEAK` 155.

11 TI MSP430 Specific Interfaces

The TI® MSP430® is a favorite micro-controller for ultra-low power applications. It has good power consumption numbers and very low sleep current draw. Although the architecture is rather old, it is clean and orthogonal and has good code density if somewhat less than spectacular execution cycle efficiency.

The compiler tool chain for the MSP430 is somewhat limited. The TI CCS offering is not strong and it seems that the IAR® Embedded Workbench for the MSP430 is the best choice for compiler. That is our target here.

11.1 MSP430 Critical Sections

The MSP430 has fixed interrupt priorities for its peripherals. Although it is possible to allow nested interrupts, we avoid such complications here. Critical sections are implemented using compiler intrinsic functions to globally disable and enable interrupts.

136 ⟨*msp430 critical section* 136⟩≡ (156)

```
static inline
void
initCriticalSection(void)
{
}
static inline
void
beginCriticalSection(void)
{
    __disable_interrupt() ;
}
static inline
void
endCriticalSection(void)
{
    __enable_interrupt() ;
}
```

Defines:
`inline`, used in chunk 101b.

11.2 MSP430 Timing Interfaces

Unfortunately, the MSP430 has a rather limited set of independent timer peripherals. The timers are rather sophisticated in what they can do, it is just that there are typically only two independently clocked timers. Most applications will end up using all the timers in some way. So, it is very possible that the timer code given here will have to be modified for your particular system. Because of this, we will split out a platform specific file for the MSP430, unlike the other platforms. Since, the compiler doesn't support *weak* declarations, it will be easier to make modifications if the MSP430 platform specific code is in a separate file.

The supplied code should give you the idea of how to interface the MSP430 to your particular timer usage. Here we will use Timer A and the CCR 0 for timing delayed events. For delayed event timing, we will clock Timer A off of the 32 KiHz ACLK and, by default, divide the clock down by a factor of 8. That will give us 4096 ticks per second or 244 microseconds per tick. This is sufficient resolution for delayed events.

137a ⟨*msp430 conditional compilation* 137a⟩≡ (155)

```
 * When compiling for the MSP430, the value of the symbol
 * MECH_TIMER_FREQUENCY gives the number of clock ticks
 * in one second. The default value is 4096.
```

It is necessary to set up some constants to define the conversion from clock ticks to milliseconds, the units of delayed event timing.

137b ⟨*msp430 delayed event helper* 137b⟩≡ (156) 138a ▷

```
#ifndef MECH_TIMER_FREQUENCY
#   define MECH_TIMER_FREQUENCY     4096UL
#endif /* MECH_TIMER_FREQUENCY */

#define MECH_MAX_CLOCK_TICKS     (UINT32_MAX - 1)
#define MECH_MAX_MSEC_DELAY\
        ((MECH_MAX_CLOCK_TICKS / MECH_TIMER_FREQUENCY) * 1000UL)
```

Defines:
MECH_MAX_CLOCK_TICKS, used in chunks 124b, 125a, and 138a.
MECH_MAX_MSEC_DELAY, used in chunks 125a and 138a.

Like before, we need functions to convert from rational time units to clock ticks and *vice versa*.

138a ⟨*msp430 delayed event helper* 137b⟩+≡ (156) ◁137b

```
static inline
MechDelayTime
mechMsecToTicks(
    MechDelayTime msec)
{
    if (msec > MECH_MAX_MSEC_DELAY) {
        msec = MECH_MAX_MSEC_DELAY ;
    }
    /*
     * We must avoid overflow if the requested number of
     * msecs is large.
     */
    return msec < MECH_MAX_CLOCK_TICKS / MECH_TIMER_FREQUENCY ?
            ((msec * MECH_TIMER_FREQUENCY) + 1000UL / 2UL) / 1000UL :
            ((msec + 1000UL / 2UL) / 1000UL) * MECH_TIMER_FREQUENCY ;
}

static inline
MechDelayTime
mechTicksToMsec(
    MechDelayTime ticks)
{
    return ticks < MECH_MAX_CLOCK_TICKS / 1000UL ?
        ((ticks * 1000UL) + MECH_TIMER_FREQUENCY / 2UL) / MECH_TIMER_FREQUENCY :
        ((ticks + MECH_TIMER_FREQUENCY / 2UL) / MECH_TIMER_FREQUENCY) * 1000UL ;
}
```

Defines:
`inline`, used in chunk 101b.
Uses `MECH_MAX_CLOCK_TICKS` 137b, `MECH_MAX_MSEC_DELAY` 124b 137b, and `MechDelayTime` 40.

So the functions that are supplied in a separate file are given below.

138b ⟨*msp430 timing declarations* 138b⟩≡ (155)

```
extern void sysTimerInit(void) ;
extern void sysTimerMask(void) ;
extern void sysTimerUnmask(void) ;
extern void sysTimerStart(MechDelayTime) ;
extern MechDelayTime sysTimerStop(void) ;
```

Uses `MechDelayTime` 40.

Now we tackle some of the quirkyness of timers on the MSP430. We will need a function to read the timer. One might think this is simple and uncomplicated, but alas no. Bottom line is that you need to do a *voting* type of read to make sure that the timer didn't roll from the low byte to the upper byte during the read. We just insist that one be able to read the timer some number of successive times and get the same result each time. In our case here we use three successive reads. In truth two should be sufficient, but since we have taken all the time to get into a loop, we might as well put on both our belt and suspenders. Fortunately, we don't do a lot of timer reading.

139 ⟨*msp430 timer services* 139⟩≡ (157) 140 ▷

```
static inline
uint16_t
readTimer(void)
{
#   define  SUCCESSIVE_MATCHES  3

    uint16_t cmp1Reg ;
    uint16_t cmp2Reg ;
    uint8_t cnt ;

    cnt = 1 ;
    cmp1Reg = TAR ;
    do {
        cmp2Reg = TAR ;
        cnt = cmp1Reg == cmp2Reg ? cnt + 1 : 1 ;
        cmp1Reg = cmp2Reg ;
    } while (cnt < SUCCESSIVE_MATCHES) ;

    return cmp2Reg ;

#   undef SUCCESSIVE_MATCHES
}
```

Defines:
`inline`, used in chunk 101b.
Uses `uint16_t` 141 and `uint8_t` 11a 11b 23b 23c 64a 72b.

Next we need to be able to set the timer with a value. In this particular usage of Timer A, we are running it in so called *continuous* mode. In this mode, the timer counts from 0 to its maximum value and then rolls over to 0, continuing to count. This mode of operation makes it a bit easier to use the other two Timer A Capture/Compare registers for other purposes in the system, but it means that to interrupt at a specific count in the future, one must add that future time to the current value of the timer. So we have a function to do just that. However, we don't want single ticks as timer residues. It is problematic in some MSP430 parts to set a timer compare register to one tick more than the current timer value and make sure and get an interrupt in that one tick. So we fudge an extra tick on those occasions where a delayed event request results in a single tick in the lower part of the time value.

140 ⟨*msp430 timer services* 139⟩+≡ (157) ◁139 141▷

```
#pragma type_attribute=__monitor
static void
setNextTime(
    uint16_t clockTicks)
{
    if (clockTicks == 1) {
        ++clockTicks ;
    }
    TACCR0 = readTimer() + clockTicks ;
    sysTimerUnmask() ;
}
```

Uses uint16_t 141.

There are two things to notice about this function. We use a `pragma` to request the compiler to generate a critical section around the function, so that the function executes with interrupts disabled and restores the interrupt state appropriately. This is not because we are sharing a critical data section. Rather it is to prevent an interrupt coming between when we read the timer and when we get it set to a new value. We are performing a read / modify / write on the value of Timer A CCR0 and do not want extraneous time to go by in the service of an interrupt when we are trying hard just to get the timer register set to a new value. Since the timer counts modulo 16 bits, then simply adding the number of ticks to the current timer value, ignoring any overflow turns out to be the right thing. If there is overflow, the timer compare register will match the timer itself when it rolls over from 0.

The second thing to notice is that this function leaves the timer interrupt itself unmasked. Nothing particularly unusual about that. Once the timer is set to a value, we want to insure that we are notified when the time elapses.

Now we arrive at another complication in MSP430 timing. Timer A is only 16 bits long. With 4096 ticks per second, we can only time 16 seconds with the timer. That is not sufficient range for delayed events. So we are forced to hold remaining bits of the timer in memory and track its value via software. We define a 16 bit value as the *upper* part of the timer as a variable.

141 ⟨*msp430 timer services* 139⟩+≡ (157) ◁140 142▷

```
static uint16_t clockCnts ;
```

Defines:
`uint16_t`, used in chunks 139, 140, and 143b.

Once we have decided to use Timer A and its Capture/Compare Register 0, the initialization and control of the interrupt mask is straight forward.

142 ⟨*msp430 timer services* 139⟩+≡ (157) ◁141 143a▷

```
void
sysTimerInit(void)
{
    /*
     * Stop and Clear the timer.
     */
    TACTL = TACLR ;
    /*
     * Set up clock source to ACLK and Divide down by 8.
     */
    TACTL |= TASSEL_1 | ID_3 ;
    TACCTL0 = 0 ;
    TACCTL1 = 0 ;
    TACCTL2 = 0 ;
    /*
     * Start the timer in continuous mode.
     */
    TACTL |= MC_2 ;
}

void
sysTimerMask(void)
{
    TACCTL0 &= ~CCIE ;
}

void
sysTimerUnmask(void)
{
    TACCTL0 &= ~CCIFG ;
    TACCTL0 |= CCIE ;
}
```

Starting and stopping the timer must take account of the upper part of the timer range that we are holding in a variable. To start the timer we put the upper part of the time in the variable and the lower part in the timer.

143a ⟨*msp430 timer services* 139⟩+≡ (157) ◁142 143b▷

```
void
sysTimerStart(
    MechDelayTime ticks)
{
    clockCnts = ticks >> 16 ;
    setNextTime(ticks) ;
}
```

Uses `MechDelayTime` 40.

Stopping a timer has its own complications also. First we have to account for the fact that since the timer and the processor run asynchronously, there is a race condition associated with stopping the timer and reading the remaining time. A simple check of the interrupt flag shows whether or not we lost the race.

143b ⟨*msp430 timer services* 139⟩+≡ (157) ◁143a 144▷

```
MechDelayTime
sysTimerStop(void)
{
    MechDelayTime remain ;

    sysTimerMask() ;
    uint16_t timer = readTimer() ;
    /*
     * Account that the timer might go off between when we
     * stop it and when we read it.
     */
    if (TACCTL0 & CCIFG) {
        remain = clockCnts ?
                (MechDelayTime)(clockCnts - 1) << 16 : 0 ;
    } else {
        remain = TACCR0 - timer ;
        remain += (MechDelayTime)clockCnts << 16 ;
    }
    return remain ;
}
```

Uses `MechDelayTime` 40 and `uint16_t` 141.

Finally, we need an interrupt service routine for the Timer A CCR0. On the MSP430, this has its own interrupt vector. Other Timer A interrupt sources share a different vector. The only complication here is to account for the upper part of the virtual timer that is held in a variable. The total count was set up so that the fraction was expired first and then all the whole 16 bit parts are counted off. This works well since by far the most frequent delayed event times fit in 16 bits worth of timer ticks.

Because of the way the MSP430 controls low power mode, it is necessary for an interrupt service routine to explicitly request that low power mode be off when the interrupt service routine finishes. Otherwise, the processor will go right back to sleep. This is a bit different from other processors that usually keep running after an interrupt occurs and software then puts it back to sleep. In this case we use it to some advantage in that we only request exiting low power mode in that case where we have expired the timer and not just counted part of the time. Given that we have to hold part of the timer value in a variable, this is convenient.

144 ⟨*msp430 timer services* 139⟩+≡ (157) ◁143b

```
#pragma vector=TIMERA0_VECTOR
__interrupt
void
timerA0ISR(void)
{
    if (clockCnts == 0) {
        MechDelayTime next = mechTimerExpireService() ;
        if (next) {
            sysTimerStart(next) ;
        }
        __low_power_mode_off_on_exit() ;
    } else {
        /*
         * Loading the compare register with the timer
         * value causes us to count an entire 16 bits worth
         * of ticks.
         */
        TACCR0 = TAR ;
        --clockCnts ;
    }
}
```

Uses `MechDelayTime` 40.

11.3 MSP430 Async Execution Interface

The MSP430 has a very simple architecture. Unlike the Cortex-M3, there is no exception level associated with executing sync functions. So it is only necessary to queue to sync function.

145a ⟨*msp430 async interface* 145a⟩≡ (156) 145b ▷

```
SyncParamRef
mechSyncRequest(
    SyncFunc f)
{
    return syncQueuePut(f, true) ;
}
```

Uses `SyncFunc` 97b and `SyncParamRef` 97a.

The non-fatal version is also very simple to implement.

145b ⟨*msp430 async interface* 145a⟩+≡ (156) ◁ 145a

```
SyncParamRef
mechTrySyncRequest(
    SyncFunc f)
{
    return syncQueuePut(f, false) ;
}
```

Uses `SyncFunc` 97b and `SyncParamRef` 97a.

11.4 MSP430 Suspending Execution

Applications will want to control the power mode, so we place the current power mode in a variable. You might think we should put some function in to *hide* the type of this variable, but that verges on overdone. If you really need fine control over the power mode you will need to write quite a bit more code to keep track of which peripherials are busy using which of the system clocks.

145c ⟨*msp430 suspend execution* 145c⟩≡ (156) 146a ▷

```
unsigned char lowPowerMode = LPM0_bits ;
```

Defines:
`lowPowerMode`, used in chunk 146a.

The MSP430 keeps the power mode bits right in the main status register and power modes are entered or exited by direct manipulation of the contents of status register. So to go to sleep, one simply sets the correct bits in the status register, making sure that the general interrupt enable is also set. Otherwise, there is no wake up from the sleep save a reset.

146a ⟨*msp430 suspend execution* 145c⟩+≡ (156) ◁145c

```
MECH_TEST_STATIC
MECH_TEST_INLINE
void
mechWait(void)
{
    beginCriticalSection() ;
    if (syncQueueEmpty()) {
        __bis_SR_register(lowPowerMode | GIE) ;
    } else {
        endCriticalSection() ;
    }
}
```

Uses `lowPowerMode` 145c.

11.5 MSP430 Initialization

As a final quirk of the MSP430, we need to be aware of a couple of things for initialization. Oddly enough, the watch dog timer is enabled after a reset. I find it most convenient to turn it off, establish the operating environment and then enable it. Also, you will need to adjust the system clock properly for your application. The code provided assumes that the MSP430 part is new enough to store calibrated clock values in flash. Your mileage will definitely vary depending up which MSP430 part you are dealing with.

146b ⟨*msp430 external scoped functions* 146b⟩≡ (155)

```
extern void sysPlatformInit(void) ;
```

147 ⟨*msp430 initialization* 147⟩≡ (157)

```
void
sysPlatformInit(void)
{
    /*
     * Hold the watchdog timer.
     */
    WDTCTL = WDT_ARST_1000 | WDTHOLD ;
    IFG1 &= ~WDTIFG ;
    /*
     * Whack up the clock to 8 MHz using the
     * calibration data from flash.
     */
    DCOCTL = 0 ;
    BCSCTL1 = CALBC1_8MHZ ;
    DCOCTL = CALDCO_8MHZ ;
}
```

12 Source Code Organization

This literate program can produce source code for three very different platforms. Traditionally, this difference is handled by conditional compilation. The program source is often littered with preprocessor statements and it is necessary to define the correct symbols during compilation to get the desired source compiled. Unfortunately, the number of preprocessor statements can get large and this makes the source much more difficult to read as you must decide which statements the compiler will actually see. This situation can get even worse when conditional compilation is used to account for compiler differences.

One can take the attitude that since this is a literate program, the appearance of the source code derived from it is not important. That is the case up to a point. The source code produced from this program has very little of the documentation embedded in it that conventionally produced code might have. But debugging is made that much worse when there is the clutter of extraneous statements in the code. Besides there is something evil that preprocessing does by changing the text that the compiler sees to be something different than what the reader sees. Minimizing that is worthwhile.

Rather than use conditional compilation, we can use the literate programming tools to produce as many source code files as we desire. So, there will be several *roots* in this literate program, one for each platform and for each compiler that is supported. We will use the naming convention of mech-<platform>-<compiler>to name the root chunks that are meant for a particular platform/compiler combination. Once the chunk is *tangled* its file may be renamed as convenient. Note that this approach does not eliminate all the conditional compilation directives. There are still some features that are included or excluded by conditional compilation, such as tracing, that are independent of the platform and compiler.

It is convenient to keep track of the version of the literate program document in the source code.

148 ⟨*version info* 148⟩≡ (149 150 152 153 155–57)
```
/*
 * THIS FILE IS AUTOMATICALLY GENERATED. DO NOT EDIT IT.
 * This file corresponds to Version 1.8 of the STSA literate
 * program.
 */
```

12.1 POSIX Version

The posix version of the header file is named `mechs-posix-gcc.h`. We are only supporting `gcc` as a compiler.

149 ⟨*mechs-posix-gcc.h* 149⟩≡

```
⟨version info 148⟩
⟨copyright 158⟩
/*
⟨conditional compilation 80⟩
 */
#ifndef MECHS_H_
#define MECHS_H_
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdarg.h>
⟨constants 11c⟩
⟨base types 11a⟩
⟨data types 10⟩
⟨posix data types 112a⟩
⟨event data types 24⟩
⟨external scoped functions 15⟩
#ifdef MECH_SM_TRACE
⟨trace data types 81⟩
⟨trace external functions 82b⟩
#endif  /* MECH_SM_TRACE */
⟨posix external functions 110b⟩
⟨inline functions 37a⟩
#ifdef MECH_TEST
⟨external test functions 6⟩
⟨main loop components declaration 101c⟩
#endif /* MECH_TEST */
#endif /* MECHS_H_ */
```

Defines:
`MECHS_H_`, never used.

The POSIX code file is named `mechs-posix-gcc.c`. For this version, the mechanisms can be compiled with:

```
gcc -std=c99 -g -O2 -c -o mechs.o mechs.c
```

150 ⟨*mechs-posix-gcc.c* 150⟩≡

```
⟨version info 148⟩
⟨copyright 158⟩
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>
#ifndef MECH_NINCL_STDIO
#   include <stdio.h>
#endif /* MECH_NINCL_STDIO */
#include <signal.h>
#include <errno.h>
#include <sys/select.h>
#include <sys/time.h>
#include "mechs.h"
#if defined(__GNUC__)
#   define __WEAK  __attribute__((weak))
#else
#   define __WEAK
#endif /* __GNUC__ */
⟨conditional defines 101b⟩
static void mechFatalError(MechErrorCode errNum, ...) ;
⟨posix critical section 103a⟩
⟨error handling 89a⟩
⟨instance allocation helper 17⟩
⟨instance functions 14⟩
⟨event queues 26a⟩
⟨event generate 31a⟩
static void sysTimerStart(MechDelayTime) ;
static MechDelayTime sysTimerStop(void) ;
⟨delayed event helper 49⟩
⟨posix delayed event helper 105⟩
⟨delayed event queue 42⟩
⟨posix timer services 106⟩
⟨delayed event service 60⟩
⟨timer service 59⟩
#ifdef MECH_SM_TRACE
⟨event tracing 83⟩
#endif /* MECH_SM_TRACE */
⟨event dispatch 61⟩
⟨fgsync data types 97c⟩
```

```
  ⟨sync queue 97e⟩
  ⟨sync queue func 98a⟩
  ⟨posix async interface 109c⟩
  ⟨posix io data 114a⟩
  ⟨posix io interface 116⟩
  ⟨posix suspend execution 119⟩
  ⟨posix initialization 121⟩
  ⟨initialization 100c⟩
  ⟨main loop sync components 102a⟩
  ⟨main loop components 102c⟩
  ⟨main loop 7⟩
```

Uses `__WEAK` 155, `MechDelayTime` 40, and `MechErrorCode` 88.

12.2 Cortex-M3 Version

The Cortex-M3 version of the code is built and tested using the CodeSourcery builds of `gcc` for ARM processors. So the header file is named `mechs-cm3-gcc.h`.

```
152 ⟨mechs-cm3-gcc.h 152⟩≡
  ⟨version info 148⟩
  ⟨copyright 158⟩
  /*
  ⟨conditional compilation 80⟩
  ⟨cm3 conditional compilation 123b⟩
   */
  #ifndef MECHS_H_
  #define MECHS_H_
  #include <stddef.h>
  #include <stdint.h>
  #include <stdbool.h>
  #include <stdarg.h>
  #if defined(__GNUC__)
  #   define __WEAK  __attribute__((weak))
  #elif defined(__ICCARM__)
  #   define __WEAK  __weak
  #else
  #   define __WEAK
  #endif /* __GNUC__ or __ICCARM__ */
  ⟨constants 11c⟩
  ⟨base types 11a⟩
  ⟨data types 10⟩
  ⟨event data types 24⟩
  ⟨external scoped functions 15⟩
  ⟨cortex-m3 external scoped functions 134⟩
  ⟨cortex-m3 timing declarations 125b⟩
  ⟨cortex-m3 power declarations 131a⟩
  #ifdef MECH_SM_TRACE
  ⟨trace data types 81⟩
  ⟨trace external functions 82b⟩
  #endif  /* MECH_SM_TRACE */
  ⟨inline functions 37a⟩
  #ifdef MECH_TEST
  ⟨external test functions 6⟩
  ⟨main loop components declaration 101c⟩
  #endif /* MECH_TEST */
  #endif /* MECHS_H_ */
```

Defines:
  `MECHS_H_`, never used.
Uses `__WEAK` 155.

The code file for the Cortex-M3 version is named `mechs-cm3-gcc.c`.

153 ⟨*mechs-cm3-gcc.c* 153⟩≡

```
⟨version info 148⟩
⟨copyright 158⟩
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>
#ifndef MECH_NINCL_STDIO
#   include <stdio.h>
#endif /* MECH_NINCL_STDIO */
typedef enum IRQn {
/******  Cortex-M3 Processor Exceptions Numbers **********/
  NonMaskableInt_IRQn   = -14,   /* 2 Non Maskable */
  HardFault_IRQn        = -13,   /* 3 Hard Fault */
  MemoryManagement_IRQn = -12,   /* 4 Memory Management */
  BusFault_IRQn         = -11,   /* 5 Bus Fault */
  UsageFault_IRQn       = -10,   /* 6 Usage Fault */
  SVCall_IRQn           = -5,    /* 11 SV Call */
  DebugMonitor_IRQn     = -4,    /* 12 Debug Monitor */
  PendSV_IRQn           = -2,    /* 14 Pend SV */
  SysTick_IRQn          = -1,    /* 15 System Tick */
} IRQn_Type ;
#include "core_cm3.h"
#include "mechs.h"
⟨conditional defines 101b⟩
⟨cortex-m3 critical section 122⟩
⟨error handling 89a⟩
⟨instance allocation helper 17⟩
⟨instance functions 14⟩
⟨event queues 26a⟩
⟨event generate 31a⟩
⟨delayed event helper 49⟩
⟨cortex-m3 delayed event helper 124b⟩
⟨delayed event queue 42⟩
#ifdef CM3_USE_SYSTICK
⟨cortex-m3 timer services 126⟩
#endif /* CM3_USE_SYSTICK */
⟨delayed event service 60⟩
⟨timer service 59⟩
#ifdef MECH_SM_TRACE
⟨event tracing 83⟩
#endif /* MECH_SM_TRACE */
⟨event dispatch 61⟩
```

```
⟨fgsync data types 97c⟩
⟨sync queue 97e⟩
⟨sync queue func 98a⟩
⟨cortex-m3 async interface 129a⟩
⟨cortex-m3 suspend execution 132⟩
⟨cortex-m3 initialization 135⟩
⟨cortex-m3 power management 131b⟩
⟨initialization 100c⟩
⟨cortex-m3 main loop components 130b⟩
⟨main loop 7⟩
```

Defines:
`IRQn_Type`, never used.

12.3 MSP430 Version

The MSP430 version of the code is built and tested using the IAR Embedded Workbench for the MSP430. So the header file is named `mechs-msp430-iar.h`.

155 ⟨*mechs-msp430-iar.h* 155⟩≡

```
⟨version info 148⟩
⟨copyright 158⟩
/*
⟨conditional compilation 80⟩
⟨msp430 conditional compilation 137a⟩
 */
#ifndef MECHS_H_
#define MECHS_H_
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdarg.h>
#define __WEAK
⟨constants 11c⟩
⟨base types 11a⟩
⟨data types 10⟩
⟨event data types 24⟩
⟨external scoped functions 15⟩
⟨msp430 external scoped functions 146b⟩
⟨msp430 timing declarations 138b⟩
#ifdef MECH_SM_TRACE
⟨trace data types 81⟩
⟨trace external functions 82b⟩
#endif  /* MECH_SM_TRACE */
⟨inline functions 37a⟩
#ifdef MECH_TEST
⟨external test functions 6⟩
⟨main loop components declaration 101c⟩
#endif /* MECH_TEST */
#endif /* MECHS_H_ */
```

Defines:
`__WEAK`, used in chunks 126, 131b, 135, 150, and 152.
`MECHS_H_`, never used.

The code file for the MSP430 version is named `mechs-msp430-iar.c`.

156 ⟨*mechs-msp430-iar.c* 156⟩≡

```
⟨version info 148⟩
⟨copyright 158⟩
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>
#ifndef MECH_NINCL_STDIO
#   include <stdio.h>
#endif /* MECH_NINCL_STDIO */
#include "msp430.h"
#include "mechs.h"
⟨conditional defines 101b⟩
⟨msp430 critical section 136⟩
⟨error handling 89a⟩
⟨instance allocation helper 17⟩
⟨instance functions 14⟩
⟨event queues 26a⟩
⟨event generate 31a⟩
⟨delayed event helper 49⟩
⟨msp430 delayed event helper 137b⟩
⟨delayed event queue 42⟩
⟨delayed event service 60⟩
⟨timer service 59⟩
#ifdef MECH_SM_TRACE
⟨event tracing 83⟩
#endif /* MECH_SM_TRACE */
⟨event dispatch 61⟩
⟨fgsync data types 97c⟩
⟨sync queue 97e⟩
⟨sync queue func 98a⟩
⟨msp430 async interface 145a⟩
⟨msp430 suspend execution 145c⟩
⟨initialization 100c⟩
⟨main loop sync components 102a⟩
⟨main loop components 102c⟩
⟨main loop 7⟩
```

For the MSP430, we split out the timer and other platform specific code into a separate file. This code is most sensitive to change for a particular system peripheral usage. The platform specific code file for the MSP430 version is named `platform-msp430-iar.c`.

157 ⟨*platform-msp430-iar.c* 157⟩≡

```
⟨version info 148⟩
⟨copyright 158⟩
#include <stddef.h>
#include "msp430.h"
#include "mechs.h"
⟨msp430 timer services 139⟩
⟨msp430 initialization 147⟩
```

13 Copyright

Since this paper is a literate program, the software that may be extracted from its source is subject to the following copyright and license.

158 ⟨*copyright* 158⟩≡ (149 150 152 153 155–57)

14 Index

References

[1] Stephen J. Mellor and Marc J. Balcer. *Executable UML: a foundation for model-driven architecture.* Addison-Wesley, 2002.

[2] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML.* Cambridge University Press, 2004.

[3] Sally Shlaer and Stephen J. Mellor. *Object Oriented Systems Analysis: Modeling the World in Data.* Prentice-Hall, 1988.

[4] Sally Shlaer and Stephen J. Mellor. *Object Oriented Systems Analysis: Modeling the World in States.* Prentice-Hall, 1992.

[5] Leon Starr. *How to Build Shlaer-Mellor Object Models.* Yourdon Press, 1996.

[6] Joseph Yiu. *The Definitive Guide to the ARM Cortex-M3.* Elsevier, 2007.