MTK

Multitasker for micro controllers

by

Gérald Garon

Gélogic inc.

PRELIMINARY

Gélogic inc. 249 Sharon, Otterburn Park, Qué. J3H 1Y8

HISTORICAL NOTES

This multitasker is based on the same concept as the Mickey Mouse Multitasker (MMM) originally developed for the Walt Disney World Automated Admission Control System as a very fast real time operating system used in the turnstiles and ticket encoders. The MMM is believed to be the fastest real time executive for the Intel 8085. The key to the fast response was its macro implementation and the use of an AMD-9519A interrupt controller as the scheduler. The MMM has been rewritten to run on the iSBC 80/24 (for an industrial control application) with software scheduling to replace the AMD-9519A, and has been augmented with more convenient delay handling facilities.

A somewhat modified version of the Mickey Mouse Multitasker has been transported to the Motorola 68HC11 microprocessor as the MTK-11. Most functions have been retained except the task **kill** call. Since the MTK-11 is intended for stand alone, fixed program, dedicated application, this "reduction" is of little or no consequence. It also makes the Multitasker safer, since it is difficult to insure that all the housekeeping needed to properly terminate a task is done.

An IBM-PC version is intended primarily as a test bench for the micro controller implementation. Because DOS is not re-entrant, a pre-emptive multitasker must be used very carefully in this environment. To ease that up, the PC version can be used in a non pre-emptive way, the only requirement being to call the scheduler when a context change is allowed.

The MTK multitasker is available for the following micro controllers:

MTK-11	Motorola 68HC11
MTK-88	Intel 80188/80186
MTK-H8	Hitachi H8
MTK-SH	Hitachi SH *
MTK-166	Siemens C16x *
MTK-68k	Motorola Coldfire **
MTK-90	Atmel AVR **

* in development

** planned.

INTRODUCTION

This multitasker is purposely limited to simple mechanisms permitting a fast response to external events, efficient resource protection and quick context switching. Actual performance will vary with micro controller, due to varying suitability of instruction sets.

The multitasker has been implemented so that state diagrams can be coded almost directly. The WAIT macro syntax enabled one to write in condensed form the equivalent of a list of "on event1 go to label1, on event2 go to label2, ...". In the C implementation, the Wait call returns the number of the first non-zero semaphore of the list.

Synchronization mechanism of the multitasker is based on the semaphore. For a clear explanation of this concept, the reading of chapter 3 of A. M. Lister's Fundamentals of Operating Systems is recommended. Mutual exclusion uses an even simpler mechanism .

CONCEPTS

The concepts underlying this multi-tasker differ somewhat from the usual ones. Most multi-taskers rely on queueing mechanisms for their internal operation. This one uses a more "associative" and "parallel" approach.

TASK

Creating a task is in fact creating a new thread (defining a new stack) and calling the specified function. The scheduler then decides from the ready tasks priorities which stack (or context) is to be restored.

In most multi-taskers, the priority is nothing else than a sort key, the scheduler then picks the first element of a ready task list. In this multi-tasker, the priority of a task is defined by the position of its associated bit in a word, this is a vestige of its original implementation which used an interrupt controller as the scheduler. But it also allows more parallelism in the task processing: many tasks can be made ready in the same operation (the same Signal, for instance). The main disadvantage is the limited number of tasks that are allowed.

EVENT

An event is something that happens. It has nothing to do with the fact that nobody, some or many one(s) is (are) waiting for it, it just happens. For instance, lightning may strike a tree, nobody is waiting for that; but you can have someone waiting for the bus, or you can have twenty-six formula one drivers waiting for the green light. This is the concept, the real life, that this multitasker implements.

This behavior is modeled using the EVENT structure:

```
typedef struct
  {
  short mask;
  int ctr;
  } MTK_EVENT;
```

Each bit of `mask` is associated to a task priority. It is set if the task of this priority is waiting for this event, it is cleared when the task resume processing. Each time this event is signaled, `ctr` is incremented by one.

SEMAPHORE

The semaphore concept used in this multi-tasker also differ from the usual one. In most implementations, a semaphore has one counter which is not allowed to become negative. In this implementation, two counters are used: the EVENT counter and the SEMAPHORE counter. The SEMAPHORE is a data structure defined as:

```
typedef struct
  {
  MTK_EVENT *evn;
  int ctr;
  } MTK_SEMAPHORE;
```

A SEMAPHORE is linked to an EVENT through the **evn** pointer. Many SEMAPHORE's can be linked to the same event, so that many tasks can use the same event independently. While the EVENT **ctr** counts the number of occurrences of the event, the SEMAPHORE **ctr** counts the number of consumed occurrences of the linked event ***evn**.

A task issuing a Wait on a SEMAPHORE will resume execution when the EVENT counter is higher than the SEMAPHORE counter, it will be suspended otherwise, including the case where the EVENT **ctr** is less than the SEMAPHORE **ctr**. The significance of this (equivalent to a negative semaphore value) is that many occurrences of the events will be required before the tasks resume execution.

The above explanation assume that the counters have infinite range, which is obviously not true. The comparison is actually performed by subtracting the unsigned SEMAPHORE counter from the unsigned EVENT counter and considering the result as a two's complement signed value. This technique takes care of the counter wrap around. One should keep in mind that the number of events that can be stacked is not infinite and is limited to +32767 and -32768 for a 16 bit counter.

RESOURCE

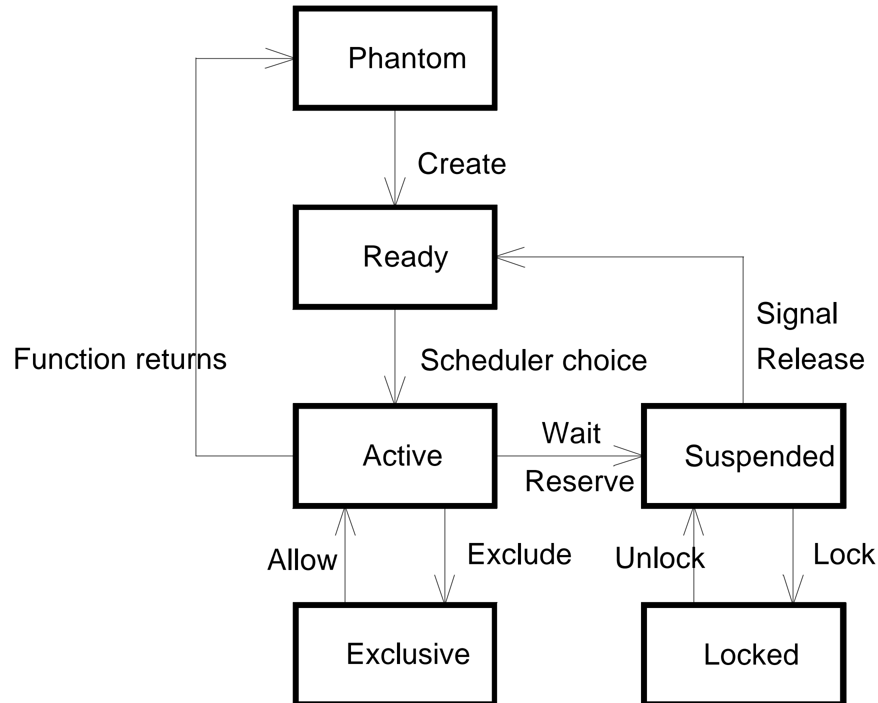
The resource is a simplification of the EVENT-SEMAPHORE combination. The RESOURCE structure is defined as:

```
typedef struct
  {
  short mask;
  short owner
  } MTK_RESOURCE;
```

Each bit of **mask** is associated to a task priority. It is set if the task of this priority is waiting for this resource. When the resource is freed, the highest priority waiting task is made active and its priority bit is cleared from the mask. The owner is zero when the resource is free, or contains the owner task bit when busy.

TASK STATES

The task state diagram indicates the various states a task may assume and which multitasker calls may cause state transitions. A description of each state follows.



Task state diagram

Phantom:	the task resides somewhere in memory (EPROM or RAM) but has not been created.
Ready:	the task has been created, has been assigned a software interrupt channel, the reason for its suspension has been cleared and/or the lock on it has been removed; it will become active as soon as no task of higher priority is active.
Active:	the task is currently executing
Suspended:	the task is waiting for some event(s) to resume execution
Locked:	execution of the task has been inhibited by another task.
Exclusive:	the task cannot be preemted by any other task, regardless of priority; note that the task of highest priority is normally exclusive and while a task is exclusive, the external interrupts are still allowed.

C CALLING CONVENTIONS

The multitasker services are available in the form of C function calls. The header file "MTK.H" must be included in all modules using the multitasker services.

MTK

Start

```
void MtkStart( void *BackgroundData )
```

This function initialize MTK and should be called before any other MTK function. The parameter BackgroundData specifies the address of the data area reserved to the background. This is the address that will be returned by the MtkTaskData function when the background is active.

VECTOR

Interrupt vector functions are used to set/reset interrupt vectors. These functions are not available on all implementations.

Set

This function places the address of the interrupt function "fct" at the vector address "vector". The new vector is installed only if the current one points to the default interrupt handler. Otherwise a MTK_VECTOR_USED error is reported.

```
int MtkVectorSet( MTK_VECTOR *vector, void (*fct)( void ));
```

Note that vector is a pointer into the vector table, not an interrupt number. If vector functions are available, the "vect.h" file contains the definition of the interrupt vector table.

Reset

This function places the address of the default interrupt function at the vector address "vector". Resetting the vector is authorised only if the currently installed vector points to "fct". Otherwise a MTK_VECTOR_USED error is reported.

```
int MtkVectorReset( MTK_VECTOR *vector, void (*fct)( void ));
```

TASK

Create

```
int MtkTaskCreate( void (*task)( void *parm ),
      void *parm, int priority, unsigned int stack_size );
```

The parameter "task" is a function address. Tasks are written as functions having one parameter. There is no "exit" call, since this is implemented through normal function return.

The parameter parm is a pointer passed to the task. Typically, this will be a pointer to a private data area. This is useful to create several copies of the same task.

The parameter "priority" specifies the priority desired for the task, 0 is highest, 15 lowest.

The parameter "stack_size" specifies the number of bytes required for the task stack. This should be the total stack required by the task itself, plus n times the stack frame size, where n is the number of nested interrupts that are allowed.

A task is terminated, and its priority and stack released when it returns.

Priority

```
int MtkTaskPriority( void );
```

This function returns the priority number of the current task. Since only one task can use a given priority slot, this amount to the task ID.

Data

```
void* MtkTaskData( void );
```

This function returns the pointer that was passed to the task in the "parm" parameter of the Create function. This is typically a structure of private data.

Lock

```
int MtkTaskLock( int Priority )
```

This function will prevent the specified task from becoming active. If several tasks lock the same task, they must all call Unlock to re-enable the locked task.

Unlock

```
int MtkTaskUnlock( int Priority );
```

This function unlocks a previously locked task.

Exclude

```
void MtkTaskExclude( void );
```

This function makes the current task exclusive, effectively locking all other tasks. The scheduler is inhibited, but interrupts are still enabled. This is useful to protect a sequence from higher priority tasks.

Allow

```
void MtkTaskAllow( void );
```

This function restores the normal priority scheme after Exclude has been called.

Preempt

```
MtkTaskPreempt
```

This is a macro used at the end of an interrupt routine to cause an immediate rescheduling.

EVENT

Clear

Clear the event control block

```
void MtkEventClear( MTK_EVENT *event );
```

An event must be cleared before any task can use it.

Signal

```
void MtkEventSignal( MTK_EVENT *event);
```

This call will increment the event counter and clear the suspended status of all tasks waiting for that event. An immediate context change may result.

Notify

```
void MtkEventNotify( MTK_EVENT *event );
```

This call will increment the event counter and readies all tasks waiting for that event. Notify does the same as Signal without task re-scheduling. No immediate context change will result. Notify is used in hardware interrupt routine to postpone re-scheduling.

SEMAPHORE

Initialize

Link a semaphore to an event, initialize at value

```
int MtkSemaphoreInitialize( MTK_SEMAPHORE *sema,
     MTK_EVENT *event, int value);
```

Evaluate

```
int MtkSemaphoreEvaluate( SEMAPHORE *sema);
```

This call returns the value of the semaphore (number of associated event un- consumed occurrences).

Wait

The C wait function requires a null terminated list of semaphore pointers. It will return the number of the first event occurred in the list

```
int MtkSemaphoreWait( MTK_SEMAPHORE *sema0,
     MTK_SEMAPHORE *sema1, ... , NULL);
```

Wait will suspend the calling task if none of the listed semaphores has a count value less than the value of the linked event; an immediate context change may result. Only the counter of the semaphore whose number is returned will be updated (incremented by one), so that no occurrence of any other event will be lost.

Typically, a switch statement is used with the value returned by the wait call.

Test

The Test function requires a null terminated list of semaphore pointers. It will return the number of the first event occurred in the list:

```
int MtkSemaphoreTest( MTK_SEMAPHORE *sema0,
    MTK_SEMAPHORE *sema1, ... , NULL);
```

Test will NOT suspend the calling task if none of the listed semaphores has a count value less than or equal to the value of the linked event; no context change may result; instead, a value equal to the number of semaphores in the list will be returned. Only the counter of the semaphore whose number is returned will be updated (incremented by one), so that no occurrence of any other event will be lost.

Typically, a switch statement is used with the value returned by the wait call.

Consume

```
void MtkSemaphoreConsume( MTK_SEMAPHORE *sema, int value)
```

This call adds the specified value to the semaphore counter, effectively "consuming" that number of occurences.

RESOURCE

Reset

```
void MtkResourceReset( MTK_RESOURCE *res );
```

This call is used to reset a resource to the released state, not used, nobody waiting. A resource must be resetted before anyone uses it, then should not be resetted after anyone used it.

Reserve

```
int MtkResourceReserve( MTK_RESOURCE *res, ... , NULL );
```

This call is used to create a protected code segment. The code segment can be entered only if all requested resources are available. If and only if all the resources are available, all the resources are marked busy and the task may proceed, otherwise, none of the resources is marked busy and the task is suspended.

Request

```
int MtkResourceRequest( MTK_RESOURCE *res, ... , NULL );
```

This call is used to conditionally create a protected code segment.. If and only if all the resources are available, all the resources are all marked busy and the function returns zero, otherwise, none of the resources is marked busy and the function returns a non-zero value.

Release

```
int MtkResourceRelease( MTK_RESOURCE *res, ... , NULL );
```

This call is used to release previously reserved resource.

STATUS CODES

Status codes returned by some calls:

MTK_OK	normal, indicates that the call was performed successfully.
MTK_PRIORITY_IN_USE	the specified priority level has been already assigned to another task.
MTK_INVALID_PRIORITY	the specified priority level is not in the 0..15 range.
MTK_NOT_BACKGROUND	this function is not available for the background.
MTK_VECTOR_USED	this interrupt vector is in use.

SERVICES

The MTK services are optional facilities of MTK.

Memory allocation

Malloc

```
void *MtkMalloc( size_t size );
```

This function is a protected version of the standard malloc functions.

Free

```
void MtkFree( void * );
```

This function is a protected version of the standard free functions.

Queue

The MTK multitasker provides a protected and synchronized data storage service implementing MTK_QUEUE_LIFO (last in first out, or queue), MTK_QUEUE_FIFO (last in first out, or stack), or MTK_QUEUE_SORTED (priority) storage mechanism. Queues may be protected and synchronized with the optional MTK_QUEUE_RESOURCE and MTK_QUEUE_SEMAPHORE attributes.

Unprotected queues are useful only if the same task is the supplier and the consumer, if two or more tasks are allowed to store, retrieve or remove from a queue, the MTK_QUEUE_RESOURCE option must be specified.

With the MTK_QUEUE_SEMAPHORE attribute, the consumer task will be suspended on an attempt to retrieve from an empty queue. It will be re-activated when an item is stored in the queue.

For sorted queues, the user must supply a comparison function with the following calling sequence:

```
int compare( MTK_NODE *item1, MTK_NODE *item2 );
```

This function should return a value of zero if both items are equal, greater than zero if item1 is greater than item2, less than zero otherwise.

Each element of a queue must be a structure whose first element is a of type MTK_NODE.

```
struct
  {
  MTK_NODE node;
  /* user data items */
  ...
  } QueueElement;
```

To use a queue, the user should declare a QUEUE variable and call setup before using it:

```
MTK_QUEUE queue;

int compare( void *item1, void *item2 )
  {
  /* comparison code */
  return( x );
  }

MtkQueueSetup( &queue, MTK_QUEUE_RESOURCE | MTK_QUEUE_SORTED, compare );
```

The queue is then ready for use. Nodes can be stored in, retrieved or removed from the queue.

Setup

```
int MtkQueueSetup( MTK_QUEUE *queue, uint attribute,
    int (*compare)( MTK_NODE *item1, MTK_NODE *item2 ));
```

This function is used to setup a queue.

Store

```
int MtkQueueStore( MTK_QUEUE *queue, MTK_NODE *item );
```

This function is used to store an element in the queue. The function returns MTK_OK if successful.

Retrieve

```
NODE *MtkQueueRetrieve( MTK_QUEUE *queue );
```

This function retrieves the first element from the queue. If the queue is empty, the function will return NULL, unless the MTK_QUEUE_SEMAPHORE attribute has been specified, in which case the calling task will be suspended until an item is stored in the queue.

Remove

```
void *MtkQueueRemove( MTK_NODE *item );
```

This function removes the specified element from the queue. This element may be anywhere in the queue.

Timer

The Tic and Sec events generated by the standard MTK timer driver are sufficient for most applications. However, some timing intensive application may require more flexible and more efficient timer implementation.

The extended timer services are implemented as an alternate timer driver. It still provides the Tic and Sec events, but also provide a timer queue.

Set

```
MTK_TIMER *MtkTimerSet( MTK_TIMER_MODE mode,
     int interval, MTK_EVENT *event );
```

This function is used to set a timer. Mode can be MTK_TIMER_ONE_SHOT or MTK_TIMER_PERIODIC. One shot will signal the specified event when time interval has expired. Periodic will repeatedly signal the event at period specified by interval

Cancel

```
void *MtkTimerCancel( MTK_TIMER *timer );
```

This function will cancel a previously set timer.

DRIVERS

Drivers are specific to the actual processor implementation, they are documented in the processor specific documentation.

Each MTK implementation attempts to provide a complete set of drivers that will make it easy to exploit the target processor resources. For the particular situation not covered (e.g. external peripherals) the supplied drivers can serve as examples.

New drivers of increasingly higher level are in continuous development.

USING MTK

Writing a task

The MTK tasks should be written as single parameter functions of the following prototype:

```
void task( void *parm );
```

The task terminates when it returns, there is no other mechanism to terminate a task. The multitasker will then free the priority number and the memory reserved for the stack. Before returning, a task must do all the required housekeeping, specifically, it must release any shareable resource, since failure to do so would lock out any other task requiring this resource.

The main criterion that must be used to determine if a task must be created is **concurrency.** Whenever two or more processes require small chunks of CPU time to perform their work, they are good candidates for multitasking.

Priority levels must be well assigned to achieve good performance. The general rule is to assign a low priority to CPU intensive tasks and high priority to sporadic, fast executing task. The CPU should trigger as quickly as possible operations that will execute without its further intervention, so that the maximum amount of work gets done.

Events and semaphores

Events and semaphores are used in a somewhat unusual way in this multitasker. This is the key to the MTK flexibility and ease of use.

EVENTs are data structures associated to some external ... events. These events are the results of some action from the outside world: a keystroke, character received on communication port, completion of a process by another task, etc. The EVENT has the capability to count occurrences, and remember which tasks are waiting for it. Many tasks may use the same EVENT dependently or independently, without even being aware of it. It is mandatory that an event be initialized before any task uses it; it is also dangerous to re-initialize it after any task has used it. EVENTs are typically declared global and initialized before tasks are created.

SEMAPHOREs are associated to the consumer of events, typically a task. They have the capability to be linked to an EVENT, and to count consumptions of events. They must be initialized before any other call involving them is issued.

EVENTs and SEMAPHOREs can be used to implement synchronization, mutual exclusion, resource protection and other multitasking mechanisms.

Wait and Signal

Wait and Signal are the basic calls for event and semaphore use. In a typical situation, the EVENTs are initialized at the beginning of the program and the tasks are created. Then the tasks will initialize their semaphores, usually with a value of zero. Zero actually means that the SEMAPHORE counter will be made equal to the linked EVENT counter. This means that there is no unprocessed occurrence of the event. If a Wait is issued in this situation, the calling task will be suspended. In suspending the task, the Wait function will store in each EVENT data structure associated with a SEMAPHORE in the list, an indication that this task is waiting for an occurrence of that event. When the event gets signaled, the EVENT counter is incremented and all the tasks waiting for it are made ready. While resuming execution, the Wait will increment the SEMAPHORE counter, to account for the event consumption, and remove the task waiting indication in the EVENT structure.

Synchronization

Synchronization in a server-consumer relationship is the most simple use of the Wait and Signal function pair. In this situation, the server is usually started first, since it would normally declare and initialize the EVENT.

```
MTK_EVENT event;
server( void )
  {
  InitEvent( &event );
  /* The typical task will loop forever */
  while ( 1 )
    {
    /* The task itself waits for something */
    ...
    Signal( &event );
    }
  }
```

Then the consumer is started, initializes its semaphore and issues a Wait on that semaphore.

```
consumer( void )
  {
  MTK_SEMAPHORE semaphore;

  InitSemaphore( &semaphore, &event, 0 );
  while ( 1 )
    {
    Wait( &semaphore, NULL );
    /* Do whatever is required... */
    ...
    }
  }
```

In this situation, once the server and consumer tasks are created, the consumer will run whenever "event" is signaled to indicate that there is something to process. If it is of higher priority than the server, it will run as soon as the Signal is executed, otherwise it will remain idle until the server issues a Wait.

Mutual exclusion

Some resources, such as peripheral devices, co-processors, buffers, cannot be used simultaneously by many tasks. They may however be shared provided some "protocol" is used to determine how the device will be shared. Typically, one should be allowed to complete its operation before another is permitted to use the resource, otherwise, strange behavior may result. For instance, if two tasks are outputting data to the screen on a character per character basis, the display will be garbled, especially if some of the characters are control codes. The mechanism used to prevent this called mutual exclusion.

Mutual exclusion is more efficiently implemented using the RESOURCE data type

```
MTK_RESOURCE scarce;

InitResource( ... )
  {
  MtkResourceInit( &scarce );
  /* Other initialization, if needed
  }

UseResource( ... )
  {
  MtkResourceReserve( &scarce, NULL );
  /* execute code using resource */
  MtkResourceRelease( &scarce, NULL )
  }
```

Mutual exclusion can be applied to several resources simultaneously:

```
MTK_RESOURCE scarce, plenty, few;

ResetResources( ... )
  {
  MtkResourceReset( &scarce );
  MtkResourceReset( &plenty );
  MtkResourceReset( &few );
  /* Other initialization, if needed */
  }

UseResources( ... )
  {
  MtkResourceReserve( &scarce, &few, &plenty, NULL );
  /* execute code using resource */
  MtkResourceRelease( &scarce, &few, &plenty, NULL )
  }
```

MtkResourceReserve will not reserve any resource unless all desired resources are available. This is sometime necessary to prevent deadlock.

Any function can be protected by wrapping it in a protected code segment. This can be done using macros.

```
#define MTK_MALLOC( ptr, size_t size )\
  MtkResourceReserve( &ResMemory, NULL );\
  ptr = Malloc( size );\
  MtkResourceRelease( &ResMemory, NULL );

#define MTK_FREE( ptr )\
  MtkResource( &ResMemory, NULL );\
  ptr = Free( ptr );\
  MtkResourceRelease( &ResMemory, NULL );

MTK_RESOURCE ResMemory;
void *ptr;

ResetMemoryResource( ... )
  {
  MtkResourceReset( &ResMemory );
  /* Other initialization, if needed */
  }

UseResources( ... )
  {
  MTK_MALLOC( Ptr, 1000 );
  /* execute code using resource */
  MTK_FREE( Ptr )
  }
```

Mutual exclusion can be also implemented using Wait and Signal, as shown in the following code fragment.

```
MTK_EVENT done;
MTK_SEMAPHORE available;

InitResource( ... )
  {
  MtkEventClear( &done );
  MtkSemaphoreInitialize( &available, &done, 1 );
  /* Other device required initializations */
  }

UseResource( ... )
  {
  MtkSemaphoreWait( &available, NULL );
  /* Device service routine */
  MtkEventSignal( &done );
  }
```

The initialize routine will setup typically one pre-occurrence of the event, so that the first task calling UseResource will be granted the resource. Subsequent callers will be held off until the first one as completed the process and issued the Signal "done". Notice that in this case, many tasks use the same eventdependently, since they share the same semaphore.

A more general case where the EVENT-SEMAPHORE pair <u>must</u> be used is when a pool of resources is shared among tasks. For instance, a pool of n buffer may be setup. so that n task may request a buffer a obtain it, once buffers are exhausted, further requesting task must be suspended.

```
#define N 3
#define BUFSIZE 80
MTK_EVENT Done;
MTK_SEMAPHORE Available;
struct TBuffer
  {
  char *next;
  char buffer[BUFSIZE]
  } BufferPool[N];
TBuffer *FreePool;

InitResource( ... )
  {
  /* Setup buffer pool */
  ...
  MtkEventClear( &Done );
  MtkSemaphoreInitialize( &Available, &Done, N );
  /* Other required initializations */
  }

UseResource( ... )
  {
  MtkSemaphoreWait( &Available, NULL );
  /* Get buffer from free list */
  /* Service routine */
  /* Return buffer to free list */
  ...
  MtkEventSignal( &Done );
  }
```

Delays

An easy way to implement delays is to use a real-time clock interrupt to periodically signal an event (Tick, Sec).

```
extern MTK_EVENT TickEvent, SecEvent;
MTK_SEMAPHORE Tick, Sec;
...
MtkSemaphoreInitialize( &Tick, &TickEvent, -19 );
MtkSemaphoreWait( &Tick, NULL );
```

If the periodic interrupt occurs every 50 ms, this will result in a one second delay. A negative value means that the related event must occur value+1 times before the task is allowed to exit the wait state.

Using events in pairs

It is sometimes required to decide which action is to be executed depending on which event occurs first. As an example, let's consider two pulses, one taken as the indication that a rotating device has completed one turn, and the other as the reference to which the rotating device needs to be synchronized. The action to be taken is to increase the speed when the pulse is late, and decrease it when it is early.

```
extern MTK_EVENT RefEvent, PulseEvent;
MTK_SEMAPHORE Ref, Pulse;
...
MtkSemaphoreInitialize( &Ref, &RefEvent, 0 );
MtkSemaphoreInitialize( &Pulse, &PulseEvent, 0 );
while ( 1 )
  {
  switch( MtkSemaphoreWait( &Ref, &Pulse, NULL ))
    {
  case 0:
    /* Increase speed */
    MtkSemaphoreConsume( &Pulse, 1 );
    break;
  case 1:
    /* Decrease speed */
    MtkSemaphoreConsume( &Ref, 1 );
    break;
    }
  }
```

The consume statements insure that both events are used up on each turn, so that the relative timing between pulse is maintained.

SPECIFIC IMPLEMENTATION ISSUES

MTK-11

The MTK-11, for Motorola 68HC11 family, is available for use either with the Motorola Cross C compiler for MS-DOS (a product no longer available) or with the ImageCraft 68HC11 C Compiler.

The MTK-11 supports 8 tasks. The EVENT and SEMAPHORE counters are 16 bit values so that the counter difference must not exceed 32767 for proper operation.

MTK-88

The MTK-88 for the Intel 80188/80186 family is implemented for MicroSoft C 7.0 - 8.0, Borland C++ 3.0 - 4.52, Watcom C/C++ 10.0 -11.0, in the SMALL, MEDIUM, COMPACT and LARGE memory models and for Turbo Pascal 6.0 - 7.0. The Microsoft implementation requires the far stack library modification to run in compact and large models.

The MTK-88 is compatible with all memory models, provided the MTK88.C source is compiled in the desired model and the MTK88.H header file is included with all modules using the multitasker function calls. It is also important to stick to good programming practices. For instance, it is required to use NULL rather than 0 for null terminated pointer list; the end of semaphore list detection will fail in the huge, large and compact models because 0 is not 0L.

The MTK-88 supports 16 simultaneous tasks. The EVENT and SEMAPHORE counters are 16 bit values so that the counter difference must not exceed 32767 for proper operation.

MTK-PC

The MTK-PC for the IBM-PC (Intel x86) in real mode. Except for the drivers, the implementation is very similar to the MTK-88, for the same compilers and is subject to the same comments. This is primarily intended as a development tool for other implementations.

MTK-SH

The MTK-SH, for the Hitachi SH1 and SH2 family, is implemented for the Hitachi C Compiler and the GNU C compiler.

The MTK-SH supports 16 simultaneous tasks. The EVENT and SEMAPHORE counters are 32 bit values so that the counter difference must not exceed 2,147,483,647 for proper operation.

MTK-H8

The MTK-H8, for the Hitachi H8/300 family, is implemented for the Hitachi C Compiler and the GNU C Compiler.

The MTK-SH supports 16 simultaneous tasks. The EVENT and SEMAPHORE counters are 16 bit values so that the counter difference must not exceed 32767 for proper operation.

MTK-166

The MTK-166, for the Siemens C16x family, is implemented for the GNU C Compiler.

The MTK-166 supports 16 simultaneous tasks. The EVENT and SEMAPHORE counters are 16 bit values so that the counter difference must not exceed 32767 for proper operation.

MTK-68k

The MTK-68k, for the Motorola Coldfire, CPU32 and 68xxx family of micro controllers is implemented for the GNU C Compiler.

The MTK-68 supports 32 simultaneous tasks. The EVENT and SEMAPHORE counters are 32 bit values so that the counter difference must not exceed 2,147,483,647 for proper operation.

MTK-90

The MTK-90, for the Atmel AVR family, is implemented for the Atmel C compiler.

The MTK-90 supports 8 simultaneous tasks. The EVENT and SEMAPHORE counters are 16 bit values so that the counter difference must not exceed 32767 for proper operation.