Date: 1 April, 1982

From: Robert S. Coren

Subject: Timers in Ring Zero MCS

To: MTB Distribution

Abstract

Various MCS applications, such as the HASP multiplexer and the Hyperchannel multiplexer interface being implemented for ASEA, have a need for some sort of timer facility in the ring zero MCS environment, to implement timeouts required by various communications protocols. This MTB describes an implementation which satisfies all these requirements, permitting an arbitrary number of timers for any channel or channels. The timers provided are reliable, in the sense that race conditions are prevented by the facility, rather than requiring user code to explicitly avoid races.

Send any comments to the author, via the Communications meeting on System-M:

>udd>Multics>Coren>mtgs>Communications (comm)

Interface:

The MCS timer facility provides the ability to schedule an arbitrary number of timers for any channel. When a particular timer comes due, a new type of MCS interrupt, the TIMER interrupt, is sent to the channel which requested the timer. A 36 bit data word is delivered with the interrupt (as the first half of the 72 bit interrupt_info value), the contents of which were specified when the timer was set. The data word can be used to identify which of several timers has come due.

Any MCS channel may have an arbitrary number of MCS timers associated with it. Each timer has a fixed bin (35) timer ID, which must be unique among the timers set for that channel. The timer IDs must only be unique per channel; different channels may have timers with the same ID.

MCS timers are reliable, in the sense that they obey a strict set of rules, unlike ordinary process timers (timer_manager_), which are subject to various race conditions, which the user procedure must guard against explicitly. If used in accordance with the rules, MCS timers can be used without any race conditions. MCS timers are guaranteed to obey the following rules:

1) An MCS timer, once set, will always deliver exactly one MCS interrupt, unless the timer is explicitly reset.

2) The interrupt for a timer will be delivered as promptly as possible, at some time no earlier than the due time of the timer. Latency is discussed in detail below.

3) Resetting a timer whose interrupt has not yet been delivered, regardless of the due time of the timer, will guarantee that no interrupt is delivered for that instance of the timer.

4) Changing the due time of a timer whose interrupt has not yet been delivered, regardless of the former due time of the timer, will guarantee that the timer interrupt for the timer arrives no earlier than the newly set due time.

MCS timer interrupts are delivered as promptly as possible. The interrupts are delivered by a procedure invoked by the pxss polling mechanism, so they will be sampled every time pxss is invoked. MCS timer polling is performed whenever a timer comes due, unlike most pxss polling, which is simply invoked at a fixed interval. This reduces MCS timer latency to a minimum. If the channel receiving the interrupt is locked when the timer interrupt arrives, the interrupt is queued, like any other interrupt. In general, the latency due to pxss response time and channel lock contention should be quite small-- tens of milliseconds. Metering is kept for timer latency, both average and maximum.

All the per-channel control entrypoints require that the channel be locked when they are called. Various error conditions will cause either a crash or a logged syserr message, depending on the severity of the error. It is possible, by changing a "special" tuning parameter, called "mcs_recoverable_error_severity" to cause all MCS timer errors to crash the system, for use in debugging. No facility is provided to crash individual multiplexers on error, since by their very nature, timing problems are very difficult to debug once any other error recovery has occurred.

No mcs_timer entrypoints have error code arguments, since it is not possible for an error to occur in an MCS timer operation which is both (a) caused by some error outside the control of the calling program, and (b) is recoverable, rather than causing a system crash. A recoverable error in an MCS timer call is indicated by a logged syserr message, and always indicates a programming error in the calling program. An error code return in this situation would not be of any use to the calling program, since all it could possibly do is log a syserr message itself and hope for the best.

List of MCS timer errors:

Errors which crash the system:

1) Call for a devx which is not locked by this process.
2) Locking errors on the mcs_timer lock (tty_buf.timer_lock)
3) Crawlout with MCS timer lock locked.

Errors which log a message and continue:

4) Call to set a timer with the same ID as an existing timer, or reset/change a timer with an ID that does not identify an existing timer. The call is ignored.
5) Calls to channel_manager$interrupt, interrupt_later, or queued_interrupt, which specify a timer interrupt. Timer interrupts may only be delivered through channel_manager$timer_interrupt. The call is ignored.

The following entrypoints are provided for control of timers:

mcs_timer$set

Takes a devx, timer ID, and a time. Sets a timer which will come due at the specified time for the specified channel (devx). If the channel already has a timer set with the specified timer ID, it is an error, and the call will have no effect (but see mcs_timer$change).

mcs_timer$reset

Takes a devx and timer ID, and resets the timer. There is a race condition where the timer interrupt has already occurred while the channel is locked, and been queued. This is handled by having mcs_timer$reset also check for a pending timer interrupt for that timer, and dequeue the interrupt. If the specified timer does not exist, it is an error, and the call will have no effect.

mcs_timer$change
Takes a devx, timer ID, and time. The specified timer is rescheduled for the new time. The race condition is the same as for mcs_timer$reset, and is handled the same way. It is an error if the specified timer does not exist.

mcs_timer$reset_all
Takes a devx, and resets all timers (if any) belonging to that channel, also dequeueing any pending timer interrupts.

mcs_timer$poll
Called by pxss, with no arguments, this procedure has the responsibility of delivering timer interrupts.

mcs_timer$unlock
Unlocks the timer lock. This is used only during the delivery of interrupts (see Implementation, below). This entry exists because MCS *timer* lock is managed by mcs_timer itself, rather than by tty_lock.

mcs_timer$verify_lock
Verifies the timer lock, crashing if it is held by the calling process. Called only by tty_lock$verify.

The first four entrypoints (set, reset, change, and reset_all) are declared in the include file mcs_timer_dcls.incl.pl1.

There will be metering data collected by mcs_timer on the timer facility, and displayed by system_comm_meters. The format of this data will be determined during the implementation, and specified in the final MCR. It will consist at least of call counts, call timings, and latency statistics.

Similarly, tty_dump and tty_analyze will be modified to be aware of the timer lists, and display them in some appropriate format. This format will also be determined during the implementation, and specified in the final MCR.

Implementation:

MCS timers are implemented as "timer blocks", allocated in tty_buf. There is a thread running through all timer blocks in the system, which sorts them in ascending order of due time, and there is also a thread running through all timer blocks associated with a particular channel. Each timer block has the following declaration:

```
declare 1 timer              aligned based (timer_ptr),
          2 next_timer       bit (18) unaligned,
          2 prev_timer       bit (18) unaligned,
          2 next_for_lcte    bit (18) unaligned,
          2 prev_for_lcte    bit (18) unaligned,

          2 pad              bit (18) unaligned,
          2 devx             fixed bin (17) unaligned,
          2 data             fixed bin (35),
          2 time             fixed bin (71);
```

Each timer block is six words long. The first two words are the threads for the two timer lists, the third to identify the owning channel, the fourth to contain the timer ID, and the remaining two for the due time.

The following variables are defined in the tty_buf header, and define the global state of the timer facility. Some of these variables may prove redundant in the implementation, and may be eliminated. Some sort of metering data will also be kept (number of timers, number of calls to various entries, average and maximum timer delivery lag, and whatnot). These will be specified in the final MCR, once an implementation is chosen, as will new system_comm_meters output.

tty_buf.next_timer_time        fixed bin (71)
The time (clock reading) at which the next timer is to go off.

tty_buf.next_timer        bit (18) aligned
The offset of the timer block belonging to the next timer scheduled.

tty_buf.timer_count        fixed bin
The number of currently scheduled timers (mature and otherwise).

tty_buf.timer_lock        bit (36) aligned
The lock protecting all the timer threads. It must be held for any operation which manipulates timer blocks.

tty_buf.timers_being_polled        bit (1) aligned
A bit indicating that there is a timer polling operation in progress. If mcs_timer$poll, after locking the timer lock, discovers this bit set, it just unlocks the lock and returns, to avoid interfering with the other processor doing polling.

Additionally, a bit (18) offset will be added to the lcte (lcte.timer_thread) which, if nonzero, is the offset of the first timer scheduled for that channel. Timer blocks are not sorted in chronological order on the per-lcte thread, but only on the per-system thread.

Locking:

The MCS timer lock is a wired lock, a spin lock. In the MCS locking hierarchy, it is above the LCTE locks, but below the tty_buf lock (used by tty_space_man) and the tty queue lock (used for queueing interrupts). Since it is a spin lock, all mcs_timer functions must run in a wired environment.

Locking for call-side operations is quite straightforward. The process (which must already hold the channel lock) calls mcs_timer, which locks the timer lock, adds, removes, or changes the timer block, rethreads the two timer threads and unlocks. If the call-side operation creates a timer which matures before any of the previously existing timers, pxss is informed of the updated next time for polling.

For the reset and change entries, while it has the timer lock held, mcs_timer will also check the queue list pointer for the channel to see if there are any queued interrupts, and, if so, call tty_lock$dequeue_timer (a new entry) to lock the queue lock, dequeue the specified timer interrupt (if it is present), unlock the queue lock, and return an indication of whether it found the timer.

Interrupt side locking is more complicated, and follows (roughly) the following path. The timer lock is held at all times except when a channel's interrupt entry is called. The additional protection of having the "timers_being_polled" flag is there only to insure serial delivery of the timer interrupts.

1) pxss calls mcs_timer$poll (on the PRDS).

2) Lock the timer lock.

3) If tty_buf.timers_being_polled is set, unlock the lock, and return. Otherwise, set it.

4) If tty_buf.next_timer is later than the current time, reset tty_buf.timers_being_polled, inform pxss of the next time we want to be polled, unlock the lock, and return. If this is the first time through the polling loop, it might be appropriate to log a message, as well, since polling is supposed to only happen when there are timers outstanding.

5) Dequeue the first timer, removing it from the global and per-channel timer threads. Update the timer count, next timer time, and next timer offset. Free the space the timer block occupied (keeping the data from the timer block in local storage).

6) Call channel_manager$timer_interrupt to deliver the interrupt. Give it (from the dequeued timer block) the devx, the timer data, and a bit (1) argument (timer_lock_unlocked) which it returns to indicate whether it had to unlock the timer lock to deliver the interrupt.

7) channel_manager$timer_interrupt behaves pretty much like channel_manager$interrupt, with the exception of how it handles a failure to lock the channel for interrupt. Except for this small amount of special handling, all metering and tracing is done just as for the normal interrupt entry. It calls tty_lock$lock_channel_int to try to lock the channel.

7a) If the channel could not be locked, channel_manager$timer_interrupt simply sets timer_lock_unlocked to "0"b and returns, with the interrupt having been added to the queue for that channel.

7b) If the channel could be locked, it sets timer_lock_unlocked to "1"b, and calls mcs_timer$unlock to unlock the lock. It then calls the interrupt entry for the channel. This is done to avoid locking hierarchy problems which would otherwise occur when the multiplexer, or even some submultiplexer, tried to call mcs_timer to set or change some other timers. Once the interrupt processing is finished, channel_manager returns.

8) Upon return from channel_manager, mcs_timer checks timer_lock_unlocked, and relocks the timer lock if channel_manager had to unlock it. It then proceeds back to step 4, above.

A new entry is added to tty_lock, tty_lock$dequeue_timer, which locks the interrupt queue, and removes the requested timer interrupt from the queue if it is there. The channel_manager timer_interrupt entry is the only means by which a timer interrupt may be signalled; if another interrupt entry is asked to deliver a timer interrupt, it is an error, and the call is ignored.

Entry: mcs_timer$set

This entry sets an MCS timer to come due at some future time. If the time supplied is in the past, it is not an error, and the timer interrupt will be delivered at the next possible opportunity. If the channel already has a timer set with the specified timer ID, it is an error, and no new timer is set. The channel specified must be locked to the calling process when mcs_timer$set is called.

Usage:

```
dcl  mcs_timer$set entry
     (fixed bin, fixed bin (71), fixed bin (35));

call mcs_timer$set (devx, time, timer_id);
```

Arguments:

1) devx Input
   The devx of the channel for which the timer is to be set.

2) time Input
   The time at which the timer is to come due.

3) timer_id Input
   A number identifying the timer. This can be used by a multiplexer to distinguish between several timers, each of which times out to control a different aspect of the protocol. This value is supplied as data when a timer interrupt is delivered (see "Timer Interrupts", below).

Entry: mcs_timer$reset

This entry resets an existing MCS timer. If the channel has no timer set with the specified timer ID, and there is also no queued timer interrupt for the specified timer ID on the channel, it is an error, and no timers are reset. Because the queued interrupts for the channel are also checked by this entry, it is always safe to reset a timer without having to worry about spurious interrupts later. The channel specified must be locked to the calling process when mcs_timer$reset is called.

Usage:

```
dcl mcs_timer$reset entry (fixed bin, fixed bin (35));

call mcs_timer$reset (devx, timer_id);
```

Arguments:

1) devx Input
The devx of the channel for which the timer is to be reset.

2) timer_id Input
The ID of the timer to reset, as supplied in a prior call to mcs_timer$set.

Entry: mcs_timer$change

This entry changes the maturity time of an existing MCS timer. If the channel has no timer set with the specified timer ID, and there is also no queued timer interrupt for the specified timer ID on the channel, it is an error, and no timers are changed. Because the queued interrupts for the channel are also checked by this entry, it is always safe to change the maturity time for a timer without having to worry about spurious interrupts later. The channel specified must be locked to the calling process when mcs_timer$change is called.

Usage:

```
dcl  mcs_timer$change entry
     (fixed bin, fixed bin (71), fixed bin (35));

call mcs_timer$change (devx, new_time, timer_id);
```

Arguments:

1) devx Input
   The devx of the channel for which the timer is to be reset.

2) new_time Input
   The new time at which the timer is to become mature. It may be either before or after the existing value.

3) timer_id Input
   The ID of the timer to change, as supplied in a prior call to mcs_timer$set.

Entry: mcs_timer$reset_all

This entry resets and and all timers and pending timer interrupts for a channel. It is not an error if the channel has no timers or pending interrupts. It can be used when it is necessary to reset the timers for a channel to a known state, such as at multiplexer crash time or whatever. The specified channel must be locked when this entry is called.

Usage:

```
dcl  mcs_timer$reset_all entry (fixed bin);

call mcs_timer$reset_all (devx);
```

Arguments:

1) devx Input
   The devx of the channel for which all timers are to be reset.

Delivery of timer interrupts:

When a timer interrupt is delivered to a multiplexer, the multiplexer's interrupt entry is invoked with an interrupt_type parameter with the value TIMER (declared in mcs_interrupt_info.incl.pl1), and a 72 bit interrupt_data value which overlays the following structure, also declared in mcs_interrupt_info.incl.pl1:

```
declare 1 timer_interrupt_data,
          2 timer_id            fixed bin (35),
          2 pad                 bit (36) aligned;
```

The timer_id element in the interrupt data is the ID of the timer which has gone off; a timer ID is supplied in the original call to mcs_timer$set. The timer ID can be used to distinguish between different timers implementing different aspects of the protocol.